THE UNIVERSITY OF MICHIGAN

Technical Report 25

CONCOMP

January 1970

A PROGRAMMING SYSTEM FOR THE SIMULATION OF CELLULAR SPACES

Ronald F. Brender

THE UNIVERSITY OF MICHIGAN

Technical Report 25

A PROGRAMMING SYSTEM FOR THE SIMULATION OF CELLULAR SPACES

Ronald F. Brender

CONCOMP: Research in Conversational Use of Computers
F.H. Westervelt, Project Director
ORA Project 07449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050,
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

January 1970

ACKNOWLEDGEMENTS

The generosity and support of many individuals and institutions have given me substantial assistance during the course of this work. I extend my appreciation of Professor John Holland, whom I value as a close friend, for serving so loyally as my Chairman; to Professor Arthur Burks for supporting this work both as a member of my committee and as Director of the Logic of Computers Group; to Professors Bernard Galler and Larry Flanigan for their interest and careful attention to so many details of the work; to the University of Michigan, the National Institutes of Health, and the Advanced Research Projects Agency for direct and indirect support in many ways; to Daniel Frantz and John Foy, long-time friends and talented system programmers, for creating much of the software foundation on which my implementation depends; to Thomas Schunior for his constant flow of ideas and techniques; to James Mortimer for his interest and courage in developing his application of the system at a time when it was still in a state of considerable flux; to Jean Slater, Bonnie Dailey, and Jan McDougall for patiently typing the many drafts; to (Mrs.) Jinx Dawson for drawing many of the figures; to Richard Laing and Thomas Dawson, capable administrative assistants, for considerable help in coping with the Establishment; and to my wife, Maurita, for her devoted support of my efforts, for her tolerance when my studies occupied my time and attention, and for her valiant struggle with my manuscript to make it stylistically acceptable. To each goes my heart felt thanks.

TABLE OF CONTENTS

LIST OF FIGURES

COMMENTARY

ON A

CONCEPT OF PLATO

In performing a computation we do not handle objects of the real world, but merely representations of objects. We are like people who live in a cave and perceive objects only by the shadows which they cast upon the walls of the cave. We use the information obtained from studying the form of these shadows to make inferences about the real world. However, we are not merely passive observers of shadows cast by real objects. We modify reality and observe the new patterns of shadows cast by the new configuration of objects. We go even further, forgetting altogether about the real objects that cast the shadows, treating the patterns of shadows as physical objects, and studying how patterns of shadows can be transformed and manipulated.

Information structures are representations of real objects just like shadows on the walls of a cave. The programmer studies how information structures can be transformed and manipulated and in doing so learns something about objects represented by the information structures. However, the real computer scientist falls in love with information structures and studies their properties not only for what they tell him about the real world but because he finds them beautiful.

- Peter Wegner
Programming Languages,
Information Structures, and
Machine Organization (1968)

Dedicated to my grandfather

Peter E. Brender
Civil Engineer and City Planner

who first introduced me to the
world of computers in 1958 when
he purchased an LGP-30 for his firm.

BLANK PAGE

ABSTRACT

A PROGRAMMING SYSTEM FOR THE
SIMULATION OF CELLULAR SPACES

by
Ronald Franklin Brender

Chairman: John H. Holland

Regular networks of similar interacting components constitute an important class of models in many disciplines, from automata theory to parallel computer systems to biological systems. Yet, no simulation system provides comprehensive facilities for studying such models conveniently by computer. Such a system is proposed and an implementation exhibited. Careful attention is given to setting forth the guiding considerations in developing the final form of the system. Primary among these is maximizing the usefulness of the system for supporting heuristic and interactive exploration of model behavior.

Chapter 1 develops the notions of cellular spaces (regular geometry, neighborhood template, transition function) and reviews models used by Von Neumann, Codd, Flanigan, Finley and Holland. Chapter 2 analyses these models and formulates the requirements for building a simulation system suitable for a wide range of cellular models. Chapters 3 and 4 describe a total programming system for simulation. A language is designed that provides novel constructs useful for cellular models. A simulation support system provides on-line monitoring of model behavior

on a graphic CRT and experimenter interaction with the system via keyboard and lightpen. Chapter 5 discusses several applications developed on the system, and evaluates and summarizes the work. Several appendices detail the implementation.

1. INTRODUCTION

John von Neumann made many contributions to the computing sciences in such diverse areas as electronic technology, computer organization, programming theory, and mathematical foundations. Two contributions, in particular, have had an enduring impact and are closely allied to efforts reported here.

Von Neumann's logical, universal space was defined to prove, for the first time, the logical possibility of a self-reproducing machine. Since that time the formalism he originated has been elaborated and extended into many application areas and taken on many guises. These include the formal models of Myhill, Yamada and Amoroso, Codd, Holland, and many others. In addition many investigations of physical phenomena, such as vibrating membranes and weather systems, and of biological systems, such as neurological networks and biological cell populations, have drawn heavily on that foundation. Formal models in these areas have been variously called cellular structures (Burks), iterative arrays (Holland, Hennie), tessellation structures (Myhill), tessellation automata (Yamada and Amoroso) and cellular spaces (Codd). We shall use exclusively the term "cellular space" as both a general name for all related concepts and for the particular models developed here.

Von Neumann's second contribution concerns the manner in which computers are used in these investigations. In surveying von Neumann's contributions, Burks [15] writes in this respect:

> The procedure which he [von Neumann] pioneered and promoted is to employ computers to solve crucial cases numerically and to use the results as a heuristic guide to theorizing. Von Neumann believed experimentation and computing to have shown that there are physical and mathematical regularities in the phenomena of fluid dynamics and important statistical properties of families of solutions of the non-linear partial differential equations involved....Von Neumann believed that one could discover these regularities and general properties by solving many specific equations and generalizing the results. From the special cases one would gain a feeling for such phenomena as turbulence and shock waves, and with this qualitative orientation could pick out further critical cases to solve numerically, eventually developing a satisfactory theory.
>
> This particular method of using computers is so important and has so much in common with other, seemingly quite different, uses of computers that it deserves extended discussion. It is of the essence of this procedure that computer solutions are not sought for their own sake, but as an aid to discovering useful concepts, broad principles, and general theories. It is thus appropriate to refer to this as the *heuristic use* of computers.... [When the computations are compared with experimental data] the heuristic use of computers becomes simulation.

Many investigations of these systems cannot be carried out without computer assistance because:

1) the behavior of these systems cannot be expressed in closed analytic form. The only way to determine the state of a system at a time t' given its state at t is to calculate the successive states at t, t+1, t+2,..., t'-1, t'.

2) The systems to be simulated must consist of at least several hundred entities to provide sufficient structure to be interpretable as representing interesting behavior.

3) The kind of behavior of interest in such systems involves time scales several orders of magnitude larger than the time scale on which the system must be simulated.

4) Exactly what constitutes interesting behavior is itself not rigorously definable. Either considerable computational power must be devoted to the task of recognizing interesting behavior as well as generating it, or on-line monitoring and interaction is a necessity to permit an experimenter to use his insight to recognize the desired behavior.

In spite of the important role that cellular models have played in much research, there are no computer languages available that have the right characteristics for generating and using simulations of cellular models. We are concerned here with filling that lack by designing and implementing a simulation language and system specifically oriented toward cellular models. Equally important, we seek to create a tool that will permit a computer to be used conveniently and heuristically.

We have fulfilled our goals by 1) developing a run-time environment (or subsystem) for use in conducting cellular simulations, 2) designing a language suited to specifying the characteristics of cellular systems, and 3) implementing a compiler for that language. This work has been conducted on the computer facilities of the Logic of Computers Group.[1]

The heuristic utility of the system is accomplished in part by exploiting the high data rate and flexible graphic capabilities of the CRT display

[1] The Logic of Computers Group is a research unit within the Department of Computer and Communication Sciences of the University of Michigan. Its computer facilities include an IBM 1800 with disk bulk storage interfaced to a DEC PDP7 with graphic CRT display.

that we have available. The system is also quite flexible and modular, offering the experimenter a range of capabilities and the opportunity even to handle specific tasks in a non-standard fashion if desired. The system will, for example, allow the user to selectively monitor various characteristics of the simulated system, to halt, to save conditions, to modify the characteristics of the system, and to resume the simulation. The language provides to the user a natural and efficient manner in which to generate a simulation. Its notation resembles as much as possible the mathematical notation frequently used in describing such systems.

We emphasize that the purpose of this investigation is the design of an implementable system which others will find has utility in conducting investigations in which cellular models are a method and not an end in themselves. Just what this author means by an implementable system is undoubtedly influenced by his own experience at doing an implementation in a particular operating environment and his personal prejudices about what is important and, hence, *must* be implementable and what can be reasonably compromised in the interest of limiting the effort required to an acceptable amount. Every effort will be made to make these biases explicit where they are recognized.

The language of our system has aspects of two distinct types. The first of these is a programming language in which to express the significant computational aspects of cell state transitions. The second of these is a command language which is employed during the course of a simulation to direct the global characteristics of the simulation system. In today's large scale, sophisticated computer systems the distinction between these aspects can be made arbitrarily small through the agency of incremental

compilers, interpreted code, and dynamic loading (and unloading) of execution modules. However, since one important characteristic of any simulation system is a fast execution rate, this author feels the above distinction will remain useful for some time.

This chapter develops the notions of cellular space and related concepts and concludes with a brief review of several models, some actually implemented and some not, that have influenced this investigation. Chapter Two presents an analysis of the requirements for simulating cellular models and suggests a particular structure for a simulation system. Chapters Three and Four set forth a particular language and system specification designed to meet the requirements developed in Chapter Two. This exposition is purposefully kept as independent of the pecular hardware and system constraints under which the author implemented the described system as is practical. Chapter Five reviews the application of this language to past and current work, suggests extensions and generally evaluates the utility of the model. Several appendices detail the actual implementation accomplished by the author in the course of this research.

1.1 On Simulation

A *system* may broadly be any collection of components or elements each of which is characterized by giving its state at a given point in time. Moreover, a component may itself be a system, and hence a subsystem of the containing system. The state of a system as a whole is known if the state of each of its components is known. The state of the system as a whole is characterized by the states of its components. A succession of system states at particular chronological instants of time constitutes a *state history*, which we will also call a record of the *behavior of the system.*

A *model* of a system A is a system B[1] that purports to represent the (relevant) properties of system A. *Simulation* is the use of a model to produce (compute) chronologically a state history of that model, which is regarded as representing a state history of the modeled system.

There are two basic strategies for doing simulation: the fixed time-step method and the next-event method. In the fixed time-step method, changes in system state (events) are assumed to occur only at times which are an integer multiple of a fixed unit of time called the time-step size. At each time-step each of the elements of the simulated system are examined to determine the new state to be used as its state for the succeeding time step. In the next-event method, changes of state may occur at arbitrary points in time. At any given time, events may be "scheduled" to occur at some later point of time. The set of scheduled events is kept in a queue in chronological order. The simulation proceeds by selecting the event that is next in the queue and computing the effects of that event. (This may or may not involve scheduling further events. If the queue ever becomes empty, the system has reached a "steady state" and the simulation terminates.) The usual practice in both methods is to assume that, when multiple events occur at the same point in time, it will make no difference (for the purposes of the simulation) in what order they are actually performed by the simulator.

1.2 Informal Discussion of Cellular Spaces

A cellular space is a collection of functionally similar cells (or modules or units) which are connected to each other in a regular manner.

---

[1] For our purposes, models are usually abstractly defined systems.

Let us take for illustration a two-dimensional plane marked into squares of unit area. Let the center of some arbitrary square be called the origin and an obvious coordinate system be imagined for identifying a particular square relative to the origin. Such a system is regular in the sense that no matter where the origin is chosen, the space "looks" the same.

Each cell has associated with it a set of cells called its neighborhood, whose states may take part in determining the behavior of the central cell.[1] Typically, the set of neighbors of a cell is determined in the same manner for every cell, and this determination is often closely related to the regular manner of interconnection or topology of the space. In our example one could choose the four cells which have a common edge with the central cell as its neighbors. (See cells A and B of Figure 1.1) Another simple neighborhood is the set of eight cells whose centers are less than 2 units distance from the central cell. (See cell C in the Figure). Note that it will often be useful to consider the central cell to be in its own neighborhood.

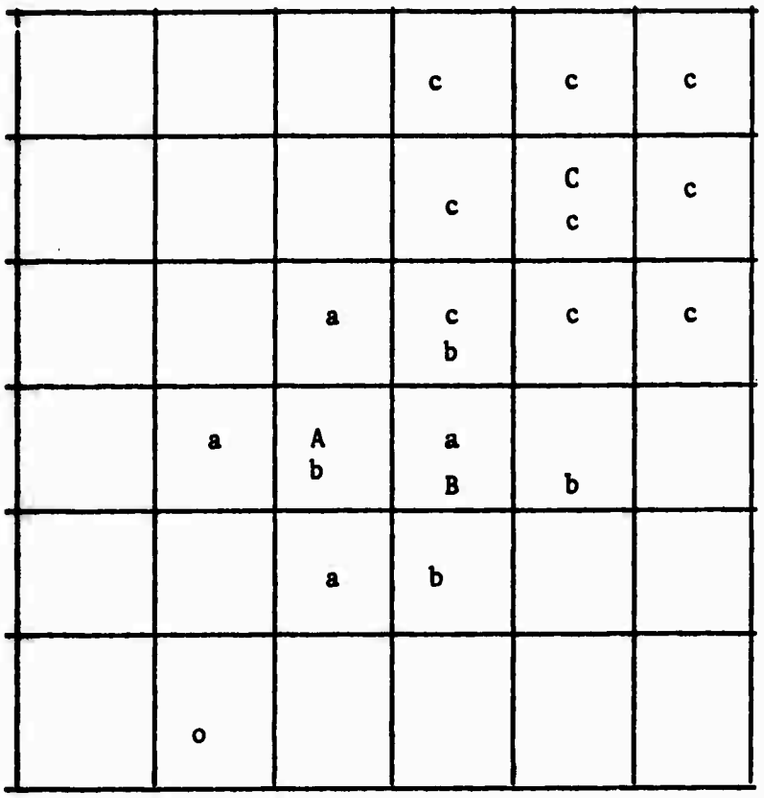
Each cell may be characterized by giving its state. Each cell in the cell space has its state drawn from the same state space[2].

The behavior of each cell is specified as a function $F: S^N \rightarrow S$

---

[1] Throughout this investigation, when speaking of a cell x and its neighborhood set of cells $N(x)$, cell x will frequently be called the *central cell* of the neighborhood. While the word "central" is often intuitively interpretable in its common sense, it is introduced more importantly to avoid an awkward naming problem.

[2] A *state space* is the set of possible states. Do not confuse this with *space state*, which is a particular state characterizing a cell space. Note that a state space need not be finite or even discrete.



o is the origin cell

Cell A has neighbors labelled "a"

Cell B has neighbors labelled "b"

Cell C has neighbors labelled "c"

Example Space and Neighborhood Templates

Figure 1.1

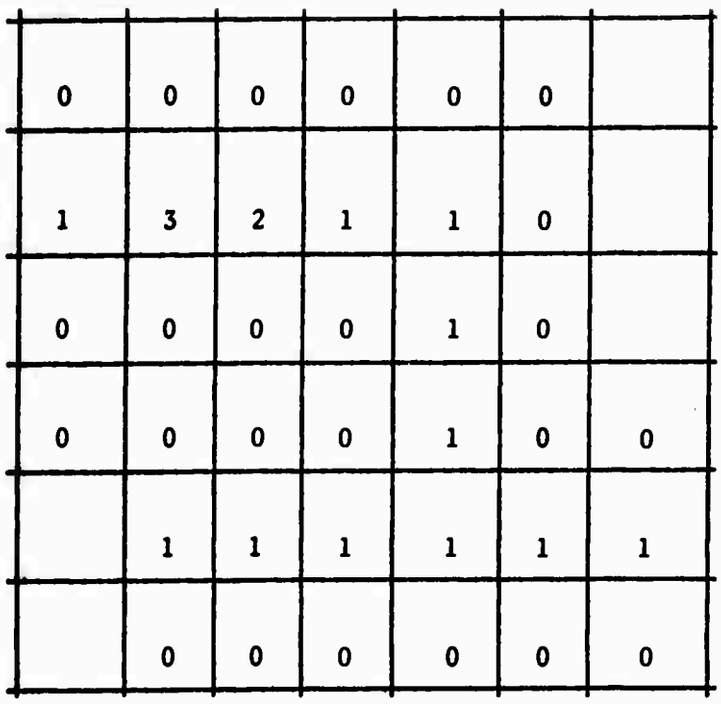
where S is the state space, and
N is the cardinality of the neighborhood set.

The arguments of F are the states of the cells in the neighborhood of the central cell at time t and the value of the function becomes the state of the central cell for the next time step, t+1. The sequence of cell states of a given cell will be called the *behavior of the cell.*

By calculating the function for every cell of the space, a new state assignment for the entire space is determined. The sequence of state assignments for the entire space is called the *behavior of the cell space.*

To illustrate how "complex" behavior may be modeled in a "simple" space, consider Figure 1.2. The space is again the two-dimensional Cartesian system and we will use as neighbors for a cell, the four cells with boundaries common to the central cell. Let each cell be in one of four possible states designated by the numerals 0, 1, 2, 3. Figure 1.2 shows a possible state assignment for a portion of the space. Also shown in Figure 1.2 is a natural language statement of a transition function. If the given transition function is applied to this space, it is easy to see that the 1 to the right of the 2 in the figure will change to a 2, the 2 to a 3, the 3 to a 1, and the rest of the space will remain unchanged. It is as though the pattern "3 2" were propagating along the path of 1's. Indeed a little reflection will satisfy one that the given transition function will allow the pattern to turn corners and to split at a junction of two paths and travel down both branches.

Note that in such a simple cell space directionality of the "signal" has been obtained by representing it with two adjacent cells. As an alternative, a more complicated cell could be specified in which direction



Transition Function:

If current state is--	And--	Then next state is--
1	Any neighbor in state 2	2
2	-	3
3	-	1
Otherwise		No change

Example Transition Function

Figure 1.2

of propagation was coded in the state of a cell. Then a signal could be represented as a single cell on a path rather than as two cells.

In cellular simulations there is a great deal of latitude in choosing between the complexity of a cell and complexity of interactions between cells. A given kind of behavior can potentially be modeled in many ways. This freedom of choice is analogous to working with thermodynamic systems; the art of the science of thermodynamics lies in deciding where to draw the boundary between system and environment in order to be able to get the information needed. Similarly, in working with cellular models there often is an art in deciding what behavior should be built into the cell state transition function and what part synthesized from groups of cells. There are many parameters to explore: the topological space itself, the neighborhood specification, the initial state assignment, the transition function.

It is not at all obvious from examination of a transition function in some cell space what range of behaviors it can be used to model. Clearly the initial state assigned to a space is crucial to the resulting behavior. Conversely, given some system that one wants to simulate with a cellular space, there is no formal approach to finding such a space. One might consider these the "proto-typical" problem statements in working with cellular systems.

1.3 Embedding and Interpreting

The concept of embedding one model in another plays an important conceptual role in the applications of cellular model simulations. Informally a *model A* may be said to be *embedded in a model B* if there exists a map

e: A → 2^B from components and their states in A to sets of components and their composite states in B which preserves the structural and behavioral properties of A in the image model B. The inverse of an embedding e: A → 2^B is an *interpretation* of B in terms of A.

Suppose M_1 and M_2 are two models. Then an embedding e of M_1 into M_2 must satisfy the following:[1]

1) Distinct elements of M_1 map into distinct sets of elements of M_2.

2) Each (input, output) of an element of M_1 maps into an (input, output) of the image element(s) in M_2.

3) If input i of element α is connected to output o of element β, then in the image e(i) is connected to e(o).

4) When a state of M_2 is the image of a state of M_1, then the behavior of M_2 does not depend on any inputs that are not in the image of the inputs of M_1.

5) The successor state(s) in M_2 of the element image must be the image of the successor state in M_1.

There are several properties of embeddings that should be noted. First, a single element in the domain model M_1 can be mapped into a set of elements of the target model M_2. This means that elements of the model M_2 may be "simpler" than elements of M_1, but by combining several elements the state space and behavior can still be represented.

Second, the image model may have "extra" elements not used in the embedding. These are no problem so long as the behavior of the elements of the image of M_1 do not depend on the behavior of elements not in the image.

Third, the regular spacial organization of cellular models permits movement to be modeled in a qualitatively unique manner. In the example of a

---

[1] Holland [11] gives a more rigorous definition of embedding in terms of compositions.

signal on a path just cited, the signal is said to move. But no thing moves, merely a pattern of states moves in a regular way. The uniformity of cellular models makes this concept of a moving pattern possible. By extending the embedding function defined earlier to be a function of time, (but still satisfying all the given constraints at any particular time), the notion of motion within a cell space is readily subsumed within the notion of embedding.

1.4 Review of Several Cellular Models

The remainder of this section will briefly review several cellular models and their applications that have had a significant impact on our work.

1.4.1 John von Neumann

John von Neumann's establishment of the concept of cellular spaces is no less important than his many other contributions to the computing sciences. He set forth a system in which behavior readily interpretable as self-reproduction could be modeled.

Von Neumann's cellular space [1, 15] consisted of an infinite two dimensional array of identical finite automata each of which may be thought of as a unit square with the aggregate covering the plane. The transition function for each cell depended on the state of the cell itself and on the state of the four "neighbor" cells which shared a common boundary with the center cell. All cells represented the same 29-state finite automata. A particular state is designated as the *quiescent* state and the transition function provides that if all five arguments are quiescent then the value of the function is the quiescent state. Von Neumann further required that the infinite array have only a finite number of non-quiescent cells at the initial time t = 0.

Briefly, certain states represented transmission states which could be combined with other such states to provide passing of an activation substate from cell to cell analogous to conduction of a signal down a wire. Certain states enable logical functions such as AND and OR to be performed on such signals. The NOT logical function was not provided for reasons connected with representing construction in the system, and hence, the logical capabilities were not functionally complete. Von Neumann synthesized the NOT function from operations concerned with returning a cell to the quiescent state; thus a signal moving down a path could be stopped (negated) by opening the path ahead of it. Certain states were associated with changing the quiescent state of neighbor cells to any of the (non-active) non-quiescent states.

Von Neumann showed how certain fundamental functional units, such as pulse encoders and decoders, could be formed by appropriate assignment of states to a contiguous group of cells. Increasingly higher level units were synthesized from these units. Finally he demonstrated how to build a tape controller and a universal constructor. This constructor is capable of reading a coded description of a cell configuration from a linear array of cells interpreted as a tape and constructing a device from the description in a nearby quiescent region of the cell space. Putting a description of the constructor itself on the tape and taking care for a few non-trivial details (e.g., a way to copy the original tape onto the newly constructed device and a way to provide the initial activation) he demonstrated a method for achieving the desired property of self-reproduction.

1.4.2 Edgar F. Codd

The von Neumann system served as a direct inspiration for Codd's dissertation [2]. Codd investigated whether the mathematical properties of universal computation and reproduction could be achieved in a cellular space simpler than that used by von Neumann.

Codd exhibited an 8-state, 5-neighbor (same neighborhood as von Neumann) space in which could be embedded a universal Turing Machine and a universal constructor interconnected to give a self-reproducing machine. The eight states are used roughly as follows:

Zero represents the quiescent state. As with von Neumann, a finite number of cells may be non-quiescent in the initial configuration, and if all neighbors of a quiescent cell are quiescent then the cell remains quiescent for the next time step. State one represents a signal path. State two represents a sheathing or insulating state that surrounds a signal path. (The concept is analogous to that of a myelin sheath in neural systems.) Codd demonstrates that any desired machine can be "built" by changing appropriate zero states to one states and then later initializing the device by propagating a special sheathing signal along the path. This signal surrounds the path by two states as it propagates. State three is used to form uni-directional paths. States zero through three are further classed as *inactive* states.

States four through 7 are termed *active* states and are used for signal states. The transition provides for conversion of one signal state into another so appropriate signals can be generated as needed.

Through heirarchical constructions working from simple gates and directional paths to a tape unit and universal constructor, Codd's develop-

ment closely mirrors von Neumann's (in spirit if not in detail) in demonstrating that his cell system has the desired properties. Codd further shows how *any* eight state-five neighbor cell space can be simulated by a two state-85 neighbor space. Hence, all of his results immediately generalize to such a two state cell space.

Of further significance here is the methodology used to *find* an appropriate eight state-five neighbor transition function. While von Neumann's approach was completely analytical, Codd's was largely empirical. Using a D.E.C. PDP-1 computer, he programmed a simple simulator to enable him to monitor the behavior of (a portion of) his state space. Successive state arrays were typed out on a typewriter. The transition function was table driven and where a previously unencountered combination occurred, the neighborhood and current state were typed out and Codd given the opportunity to define the new state to be used. By incrementally defining the transition function and then observing the behavior provided, back-tracking as necessary, Codd gradually built up a transition function which exhibited the needed behavior. Several different neighborhood configurations were tried and abandoned during this exploration.

It is important to note the heuristic and exploratory nature of this investigation. The visual pattern recognition and insight of the investigator play an important role in determining the course of the investigation. Since the number of transition functions possible is 8^{8^5} (actually fewer after considering the symmetries Codd imposed) it is clear that exhaustive search methods could not possibly be a practical approach to the problem.

1.4.3 Larry K. Flanigan

Flanigan [6] investigated the electrophysiological properties of the A-V node (atrioventricular node) of mammalian hearts, both in the laboratory and via computer simulations. He developed a cellular model of the node based on anatomical and electrophysiological data and on earlier cardiac cell models. The model was concerned with the propagation of "excitation" from the atrial edge to the ventricular edge of the node and the role this excitation plays in coordinating contractions of the heart. His simulations enable him to suggest possible mechanisms for several important classes of node behavior, both normal and pathological. And quite importantly his model supports the contention that cardiac behavior is explainable on the basis of a cellular system rather than a syncytium, i.e., as the result of the effects of local autonomous units rather than a single functional entity.

Each cell of the model is described by a single transition function determining its behavior from its own state and that of (up to) six neighbors. Cell state is described by typically 5 or 6 parameters. A two dimension, six neighbor, hexagonal geometry was used. Networks of from approximately 40 to 300 cells were simulated with configurations having either an approximately rectangular shape or a "funnel" shape. Cells along the input edge were identical with the rest of the array except that they had "neighbors" lying outside the array. The state of these external "input cells" was explicitly controlled to provide the effects of external activity.

1.4.4 Marion Finley, Jr.

Finley is one of the most recent in a line of investigators beginning with D. O. Hebb[8] and including Rochester [14], Holland [4, 14] and

Crichton [4]. These researchers have worked with various and increasingly sophisticated models of neural systems in an attempt to demonstrate the validity of the cell assembly concept postulated by D. O. Hebb. Basically, a cell assembly is a collection of neurons that comes to operate as a functional unit as a result of its stimulus history. Finley claims that he was very close to this demonstration when the desire to graduate and lack of computer funds curtailed his investigation.

His simulations typically involved 400 neurons in a two-dimensional rectangular array. The number of inputs received by each neuron (i.e. its neighborhood) ranged from 10 to 60 in various experiments. The pattern of interconnection, randomly generated at the beginning, was fixed during a single experiment. Thus each cell had an explicitly designated neighborhood. In later experiments the distribution of these interconnections was biased by the distance between two cells. Each interconnection was characterized by a "synaptic value" which determined the extent of influence of each of a cell's inputs. The behavior of these synaptic values as well as the statistical behavior of the network as a whole was studied in detail.

1.4.5 John H. Holland

Inspired in part by von Neumann's cellular space and seeking a more general formal basis for his study of parallel systems and adaptation, Holland [10] defined a class of iterative circuit computers. Iterative circuit computers are composed of a large number of identical modules operating in parallel in a synchronous fashion. Because of its compactness and strong relationship to the simulation model of this thesis, the following characterization is quoted from Holland:

The *class* of iterative circuit computers is the set of all devices (automata) specified by the admissible substitution instances of the quintuple (A, A°, X, f, P). Each particular quintuple designates a distinct interative circuit computer organization. Intuitively the five parts of the quintuple determine the following features of the organization:

1. Selection of A determines the underlying geometry of the array, particularly the dimension--thus, among other things A determines whether the array is to be planar, 3-dimensional or higher dimensional;
2. Selection of A° determines the standard neighborhood or connection scheme of modules in the array--thus A° determines the number and arrangement of modules directly connected to a given module;
3. Selection of X determines the storage register capacity of the module;
4. Selection of f determines the instruction set and related operational characteristics of the module;
5. Selection of P determines the path-building (addressing) capabilities of the modules.

More formally, the admissible substitution instances of each of the five quantities are:

1. A must be some finitely generated abelian group having a designated set of generators, say $g_0, g_1, \ldots, g_n$, with the restriction that no constraining relations involve g_0. That is, the group is free on g_0. The positions of modules in the array are indexed by elements of the subgroup A' generated by $g_1, \ldots, g_n$. The time-step is given by the exponent of g_0. Thus $\alpha = g_0^{\,t}\, g_1^{\,j1} \ldots g_n^{\,jn}$, an element of A, specifies time-step t at the module having coordinates $(j_1, \ldots, j_n)$. By choosing the subgroup A' appropriately the modules can be arranged in a plane, or a torus, or an n-dimensional array, etc. For example, if A' is free on two generators g_1, g_2, an infinite planar array is specified.

If the constraining relations

$$g_1^{100} = e$$
$$g_2^{100} = e$$

where e is the group identity, are added, a 2-dimensional torus 100 modules in each diameter (10,000 modules total) is specified.

2. A° must be a finite set of elements, $\{a_1, \ldots a_k\}$, belonging to the subgroup A' of A. For a module at arbitrary location, A° specifies the arrangement of directly connected modules. Thus the modules directly connected to the module indexed by $\alpha = g_0^t g_1^{j1} \ldots g_n^{jn}$ will be the modules at $a_i\alpha = g_0^t g_1^{j1 + k_{i1}} \ldots g_n^{jn + k_{in}}$, where $a_i = g_1^{k_{i1}} \ldots g_n^{k_{in}}$. For example, if there is a module at (j_1, j_2) relative to generators g_1, g_2, and directly connected modules are to be at coordinates $(j_1 + 1, j_2)$, $(j_1, j_2 + 1)$, $(j_1 - 1, j_2)$, and $(j_1, j_2 - 1)$, then A° should be the set $\{g_1, g_2, g_1^{-1}, g_2^{-1}\}$, where g^{-1} is the group inverse of g.
3. X can be an arbitrary finite set. The set of internal states of the module is the set $S = X \times Y$ where Y is the cartesian product $\prod_{j=1}^{k} Y_i$, $Y_i = \prod_{j=1}^{k} \{a_j \cup \phi\}$ and $a_j \in A^\circ$. That is, Y is the set of $k \times k$ matrices with entry Y_{ij} being a_j or ϕ. The set X corresponds roughly to the possible states of the module's storage register; the set Y consists of the possible gate configurations for the paths--see the transition equations below.
4. f can be an arbitrary finite function of the form
$$f : \{S \cup \phi\}^k \to S$$
f determines the instruction set, that is, the permissible transitions of the storage register states--see the transition equations.
5. P can be an arbitrary finite function of the form
$$P : S \to Y$$
P determines changes in path gating--see transition equations.

Having chosen (A, A°, X, f, P), the behavior of the corresponding iterative circuit computer is completely determined by the following state transition schema:

[$S(\alpha)$ will designate the element of S associated with α under the mapping defined recursively by the transition schema. Under interpretation $S(\alpha)$ designates the internal state of the module with space-time coordinates $(t, j_1, \ldots, j_n)$ corresponding to $\alpha = g^{t}{}_{g_1}{}^{j1}\ldots g_n{}^{jn}$. This convention will also be used for the components of S and, in particular, $Y_{ij}(\alpha)$ will designate the value of element Y_{ij} in the matrix Y associated with α. Note also, that $g_0\alpha = g_0{}^{t+1}g_1{}^{j1}\ldots g_n{}^{jn}$ designates the module at the same space coordinates as given by α, but at time $t+1$ rather than t.]

The transition schema for $Y(g_0\alpha)$ determines the path-gating at time $t+1$ in the corresponding module in terms of the internal state of the module at time t, $S(\alpha)$. Under interpretation, if $Y_{ij}(\alpha) = \alpha_j$ the gate is open so that information can be passed *without* a time-step delay *from* the module at $a_j\alpha$ through the module at α *to* the module at $a_i{}^{-1}a$; if $Y_{ij}(\alpha) = \phi$ the gate is closed. In other words $Y(\alpha)$ tells how information is to be channeled through the module to its immediate neighbors; the matrices for these neighbors tell how the information is to be sent on from there, etc. (Details of the information transfer are given by the transition equations for $S(g_0\alpha)$).

$$Y_{ij}(g_0\alpha) = Y_{ij}(\alpha) \text{ if } Q_{ij}(\alpha) = 0 \text{ and } P_{ij}(\alpha) = \alpha_j$$
$$= P_{ij}(\alpha) \text{ otherwise}$$

where $P_{ij}(\alpha)$ is the matrix element (i,j) of $P(S(\alpha))$

and $Q_{ij}(\alpha) = \Lambda_{h=1}^{k} q_{jh}(a_j\alpha)$

$$q_{jh}(\beta) = 0 \text{ if } Y_{jh}(\beta) = \phi \text{ and } P_{jh}(\beta) = \alpha_h$$
$$= 1 \text{ if } P_{jh}(\beta) = \phi$$
$$= \Lambda_{l=1}^{k} q_{hl}\,(a_h\alpha) \quad \text{otherwise}$$

where $\Lambda_{x=1}^{k} q_{jx}$ is the conjunction of the $q_{jx}{}^{0}$.

Under interpretation $P_{ij}(\alpha)$ specifies a proposed gate-setting for time $t + 1$ at the given module. $Q_{ij}(\alpha)$ prohibits any change in the gate-setting, *if* there are any changes elsewhere in the path leading through that particular gate. This prohibition prevents the following unstable situations:

1. A cycle of connections without delay (operation of modules belonging to such a cycle would in general be indeterminate).
2. An indefinitely long chain of connections without delay (otherwise a possibility in certain interesting infinite arrays).

The transition equations for $X(g_0\alpha)$ are given in terms of a function I: $B \to S$, defined for a subset B of A. Under interpretation I represents input to the computer:

$$X(g_0\alpha) = f_x(S'(a_1\alpha, 1), \ldots, S'(a_k\alpha, k))$$

where f_x is the projection of f on X

and $S'(\beta, i) = S(\beta)$ if $Y_i(g_0\alpha) = (\phi, \ldots, \phi)$

and $I(\beta)$ is not defined

$= I(\beta)$ if $Y_i(g_0\beta) = (\phi, \ldots, \phi)$ and $I(\beta) \in S$

$= f(S'(Y_{i1}(\beta)\cdot\beta, 1), \ldots, S'(Y_{ik}(\beta)\cdot\beta, k))$

otherwise

where $\phi\beta = \phi$ and $S'(\phi, j) = \phi$

Holland [9], Comfort [3] and others have discussed possible particular iterative circuit computers, including their set up, programming, etc. Holland [11] also shows that his ICC's contain a subset of composition-universal compositions. Basically this means there exist ICC's in which any machine for computing a computable function may be embedded, anywhere in the ICC.

Using Holland's formalism, we may define von Neumann's cellular space as follows:

1) A = the free group over two generators a_1, a_2.

2) $A^\circ = \{1, a_1, a_2, a_1^{-1}, a_2^{-1}\}$. Note that the group identity must be included in A° in order that the domain of the transition function include the module itself.

3) $X = \{1, 2, 3, \ldots, 29\}$

   $Y = Y_o$ where Y_o is a 5 by 5 matrix with all zero elements, i.e., no information "passes through" any cell to others.

4) $\forall s \in S$, $P(s)/_Y = Y_o$

   (weaker conditions would suffice but this is convenient.)

5) $F: S^5 \rightarrow X$ and, further, condition 4 implies that F is completely determined by F restricted to X^5.

The enumeration of F is not relevant here except to recall that von Neumann designated a quiescent state $q \in X$ and required that

1) $F(q,q,q,q,q) = q$

and

2) at most a finite number of modules be non-quiescent at time $t = 0$.

2. ANALYSIS AND FORMULATION OF SYSTEM REQUIREMENTS

The review of cellular models in the previous chapter presents a diverse and challenging variety. Yet the strongly cellular character of these models suggests an enticing commonality that we shall seek to explore and develop in this Chapter. By considering in turn each of the major characteristics of cellular spaces and by comparing the ways each has appeared in our examples, we shall work toward a single formulation suitable for a range of applications.

2.1 Discreteness

Cell spaces are discrete in time and space. Their highly parallel formulation is inherently closer to the fixed time-step method than to the next-event method. Moreover, a synchronous or fixed time-step model is considerably simpler to implement than a next-event model. (Of the examples of Chapter 1 only Flanigan used a next-event form of simulation.) Therefore, we shall assume that cellular spaces are synchronous--that is, that a new state value is calculated for every cell at the same instant based on the current state of a space, and that the resulting configuration is the state of the space at the next time step.

2.2 Uniform Connectedness

We are concerned with "regular" networks of cells distributed over space. Holland's ICC model in terms of abelian groups gives the most

general notion of regular arrangment seen in the several examples. A more general notion, in terms of finitely generated group graphs, is suggested by Wagner [16] who recognizes, however, two important reasons for restricting attention to the finitely generated abelian group graphs:

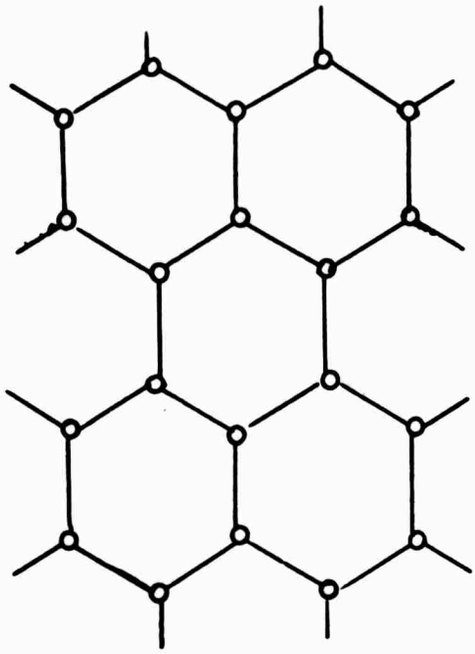
1) such groups give rise to many "nice" structures such as planes, cylinders and toroids, and
2) the theory of such groups is well developed and decidable in contrast to the general theory of finitely generated groups.

To these reasons we can add:

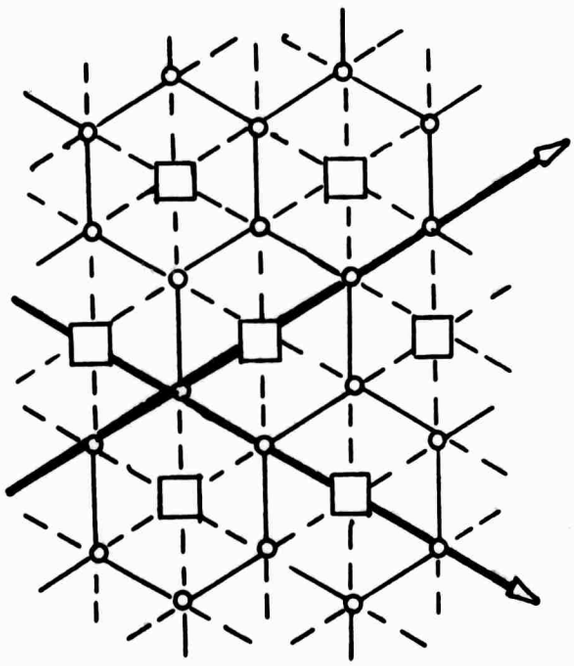
3) abelian groups give rise to a natural and physically interpretable coordinatization scheme, and
4) data structures corresponding to the abelian group generator are rather straight-forwardly implementable while those of the more general group graphs are not.

Flanigan's use of a hexagonal topology may appear to fall outside the abelian group formulation, but in fact does not. The hexagonal topology is easily obtained by choosing as neighbors those cells at coordinates N = ((1,0), (0,1), (-1,1), (-1,0), (0,1), (1,-1)) relative to the central cell. This is apparent if one imagines the major axes to be at an oblique rather than right angle. (See Figure C.2 in Appendix C.)

Yamada and Amoroso [19] give a more complex hexagonal example with cells being centered on the vertices of covering hexagons and three neighbors besides the central cell. Figure 2.1 (taken from Yamada and Amoroso) shows how to embed this topology in the two dimensional framework by using oblique axes and introducing "dead" cells at coordinate points {(3m+2+n,n)|



(a) HEXAGONAL CELL SPACE



(b) EMBEDDING IN SQUARE CELL SPACE

Figure 2.1 EMBEDDING OF HEXAGONAL CELL SPACE

m, n ε Z} and suitably extending the transition function[1]. Similar techniques may be used for other configurations.

For all of these reasons this author feels the abelian group foundation represents a good basis for a cellular simulation system. We shall take advantage of the coordinatization property and represent spaces and vectors in terms of the more intuitive coordinate notation.

2.3 Size of Simulation and Concept of Quiescence

To perform an actual simulation one is often faced with the problem of simulating a logically infinite space on a finite computer. A size must be determined and boundaries specified and even if the space is finite (and within the capacity of the computer) there is still the task of defining boundaries.

Introducing boundaries in an "infinite" model raises the problem of what to do when calculating the transition function of a cell within the boundary which has a neighbor lying logically outside of the boundary. In models such as von Neumann and Codd used there is a very natural solution: structure the cell space data management routines such that when the state of a cell outside the actual simulation is required, then the quiescent state is given. This is consistent with the formal requirement that only a finite portion of the space is non-quiescent and, hence, everything outside

---

[1] This is handled for this case as follows: The cell state space, let us call it S, is enlarged by adding a new state D (the dead state) which is assumed only by the dead cells. For a transition function F, $F(s_1, \ldots, s_7) = D$ if and only if $s_1 = D$. If $s_1 \neq D$ but $s_3 = D$ then $F(s_1, \ldots, s_7)$ is determined by s_1, s_2, s_4, and s_6 only. If $s_1 \neq D$ and $s_6 = D$ then $F(s_1, \ldots, s_7)$ is determined by s_1, s_3, s_5 and s_7 only. With such constraints one has embedded the four neighbor hexagonal space of Figure 2.1a in a cartesian seven neighbor space.

the actual simulation is quiescent and will remain so by virtue of the transition function.

A similar technique can be used in many biologically oriented models by substituting "background" for "quiescent". The simulated cells can be considered immersed in a "larger" mass of cells whose behavior is known and fixed. A straightforward extension of this idea is to permit the background state to be varied as a function of time, or to be randomly varied from access to access during the same time step, or both.

This is a powerful technique for simulating a large or infinite space and at the same time it maintains the simplicity of writing the transition function independent of concern for boundary conditions and configurations.

One service that would be very helpful for the simulator to perform is to monitor the state of cells that are "near" the boundary to detect states that would, if part of the neighborhood of a cell outside the boundary, lead to a change of state for such an exterior cell. This would allow the experimenter to detect a "spread" of non-quiescent activity beyond the point where the validity of the simulation may be questioned. This is, of course, easiest to perform in spaces having small neighborhoods (i.e. small in distance from central cell) and a well-defined quiescent or background state. The ideal solution would be for the simulation system, on detecting such a situation, to expand the boundaries of the actual simulation to include the newly "active" cell and keep going. A more realistic approach might be to allow the investigator to specify the new boundaries (subject to the same system constraints on specification as were followed originally) and then to continue with the expanded simulation.

Alternately one can avoid some edge problems by using Holland's observation that abelian spaces are easily made cylindrical or toroidal by adding certain constraints to the group. In some simulations this kind of "wrapping-around" could be more suitable than the quiescent/background approach suggested above. (A possible example is the propagation of a "wave" of activity in a neural network.) This approach also provides a simple way to "close" a finite group of cells on itself without imposing an unwarranted "stabilizing" effect. Clearly this, too, can be accomplished at the level of the data management routines without requiring any consideration at the level of the transition routine.

Both of the above may be useful where a regular neighborhood configuration is desired. The problem of edges is also reduced in finite spaces where each cell has an explicit neighborhood completely contained in the space. Finley's simulation is an example of this.

But sooner or later a boundary must be specified--if only to determine storage requirements in a computer. To this author's knowledge, no one actually doing cellular simulation, with the exception of Flanigan, has been concerned with providing boundaries more complicated than simple rectangles. In his case the boundary was a simple convex polygon. Thus, until further motivation arises, it should be sufficient to provide a facility to define a boundary for a cellular simulation as being an arbitrary convex polygon. A simple way to define a polygon in a two dimensional space is to list the coordinates of the vertices in clockwise order.

2.4 Neighborhoods

Once the cell space geometry is determined it is usually possible to give a uniform specification for the neighborhood. One can think of a fixed

template that covers a cell's neighbors when some fixed point of the template is on the central cell. This concept finds natural expression as an n-tuple of vectors in the coordinates of the underlying space. The i^{th} element of the n-tuple represents a displacement which when added to the coordinates of the central cell gives the i^{th} neighbor of that cell. A single template is defined which is applied uniformly throughout the space.

In simulations where the neighborhood must vary from cell to cell a method will be proposed for making that specification part of each cell's data structure. Further there is no reason that part of the neighborhood cannot be uniform across the space and part be cell dependent.

2.5 Transition Functions

The writing and rewriting of the transition functions used throughout an investigation is likely to require a significant amount of an investigator's effort. There is much need, therefore, to make the language for expressing transition functions both powerful and oriented toward the particular requirements of cellular simulations.

Transition functions are, by their very definition, local rules of behavior that are applied throughout the space. A central quality of a transition function language should be this "localness". Since it is a function, it will suffice to construct it as a subroutine called by the simulation executive with the states of neighboring cells as its arguments. The only characteristic of the cell geometry and neighborhood template that is essential in constructing the transition function is the number of neighbors (arguments).

2.5.1 Cell Data Structure

Each cell of a space has the same data structure which, except for the number of computer words required to represent that structure in computer storage, need not be known to the simulation system. It is not the purpose of this dissertation to develop or investigate languages with data structure capabilities as such. However, the availability of such capabilities is believed to be so useful in coding cellular simulations that basic though powerful data structure facilities will be included in the particular language exhibited in Chapter 3.

In any systematic approach to data structures an important adjunct is:

1) an ability to define operators to augment the pre-defined ones of the language,

and

2) the ability to write constants of the newly defined data types in a natural manner.

Both of these capabilities will be prominent in the language developed in Chapter 3.

2.5.2 Parameters Called by Value

The method of calculating the next state of a cell space requires the simultaneous parallel calculation of the next state of each cell of the space followed by a simultaneous and instantaneous change of each cell to that new state. The result is the new cell space state. But a simulation on any real computer (perhaps excepting Illiac IV) proceeds serially cell by cell. Clearly the next state of a cell cannot be immediately substituted

into the data structure upon calculation--the cell is potentially the neighbor of some cell whose transition is yet to be calculated.

This implies that the parameters of a transition function must be called by value and cannot be modified by the transition function itself. The value of the function is a state description which must be saved separately from the *current* cell space state until each cell's *next* state has been computed. When a new state has been computed for all cells, the next state can be considered the current state and the computation of a new next state begun.

This can be implemented by using two data areas, each sufficient to completely represent the state of the space. One area contains the current space state and the other contains the next space state while it is being computed, and the roles are reversed for the successive time steps. Although this requires twice the storage needed to simply represent the space state, it is in general the only satisfactory technique.

In some spaces with simple geometry and small neighborhoods, more specialized techniques could be employed. For example, in the Von Neumann-Codd simulations at most the equivalent of one row plus one cell need be stored in duplicate. In more complex cases, one could search for scanning algorithms that were based on the neighborhood relation in such a way to systematically complete some "compact" region of cells and expand regularly from there to eventually cover the whole space. Parnas [13] has developed some formal techniques for reducing both the space and computation requirement of simple spaces and neighborhoods. However, our desire to permit arbitrary neighborhoods, such as Finley used, and even dynamically changing neighborhoods, as in Holland's ICC's, clearly implies that nothing short of full duplication of the cell space data structure will suffice.

2.5.3 Input-Output

A language for transition functions obviously needs no input-output capabilities in the usual sense; it is simply a function. However, because some interaction with the transition function may be quite useful at various times in an exploratory simulation, some rudimentary I/O facilities are recommended.

2.6 Input to Cell Space

Inputs to a cell space are states which are not specified by the transition function but which are controlled independently outside of the cell space. It has been suggested that an additional "dummy" entry be included in the neighborhood of each cell which, by convention, is not the state of some cell but rather input for the central cell. This has the conceptual inelegance of grouping two quite different functions into the neighborhood concept. Further, all "real" neighbors of a cell have states drawn from a common state space, while the state space of the input will quite likely be different.

Yamada and Amoroso [19] formalize input as a set of possible transition functions that may be applied to the space. The input determines which transition is used on any given time step and the same one must be used for every cell in the space. This, however, does not permit modeling of spacially distributed inputs.

Yet another approach is to consider certain cells to have their states controlled externally and not by the transition function.

Because none of the above notions of input is sufficiently flexible to fulfill a variety of requirements, we have developed the following quite different approach. Consider that there are a finite number of input streams.

An *input stream* is a function of time only with values in some designated state space. Which stream is used as the input of a given cell is designated by the value of a substate of that cell's data structure. This permits input to be spacially distributed in a natural manner (even with dynamically changing spacial distributions) and the state space of the input to be conveniently distinct from that of the cell itself. Alternately, if the input state space and cell state space (except for the input designation) are the same, then the transition function can, under appropriate conditions, assign the input value as the state of the cell.

2.7 Output and Monitoring of Cell Space

In spaces with quite simple state spaces, the state is readily represented by one of a small number of distinguishable graphics. Of crucial importance is the ability to see the states of cells in their spacial arrangement. In working with a complex state space, consisting perhaps of several independent substates, it is difficult to represent the full state of many cells in a compact and intuitive manner. There is little previous experience to draw on. Our experience indicates that one does not usually need to know the full state of a group of cells; rather some characteristic property (often a particular substate) is sufficient to convey the desired information.

Such properties may be computed by a set of auxiliary functions which we refer to as *maps* or *mapping functions*. Each map is a function from the cell state space into some distinctive set of symbols.[1] These symbols are

---

[1] We admit to being motivated in part by the available hardware. The display (a D.E.C. 338) has a character generator that allows faster execution and more compact data tables than in other plotting modes. Moreover, the

chosen to be heuristically suggestive of the value of the property represented and uniform in size to permit easy presentation on CRT display.

We have adopted the strategy of having the state of a cell mapped into a "character" to be used to represent its state to the experimenter. After each update of the state of the space, the display mapping can be computed for each cell of the space, and the resulting graphics displayed in their spacial arrangement. Provision should be made for several such mapping functions, perhaps with distinct character sets, to be able to view different aspects of the space state.

2.8 Interactive Requirements

A range of commands will be required to effect control of a cellular simulation. A list of such commands could be made arbitrarily long as more and more general and powerful commands are constructed and more specialized requirements included. The following kinds of capabilities are minimal and should be included in any cellular simulation system.

1) Specify important characteristics of a cellular simulation:
   transition function
   neighborhood
   boundary
   edge convention, etc.

2) Simulate for a given number of time steps while monitoring the behavior.

3) Change the display mapping function used to designate cell state characteristics.

4) Provide general capabilities for changing the state of individual cells.

---

character set is user defined from control tables in a reserved area of memory so that a "character" may in fact be any figure that is desired. At most 128 such characters may be specified, of which one must be an "escape character mode" pseudo-character.

5) Provide a capability for interacting with the transition function itself, to change parameters, etc.

6) Provide a mechanism for a transition function to indicate a "not-defined" condition and perform interaction with user to determine what to do without upsetting the logical integrity of the simulation.

7) Save a given cell space state with appropriate identification for later possible use. The saved data should contain sufficient coordinate information for the purposes of the restore operation.

8) Restore the current cell space state from a given named file area. If boundary information differs, provide an option for keeping the parts that overlap.

9) Provide facilities to conveniently define input streams.

2.9 New Language or Old?

Available languages do not provide the kind of data structure flexibility that is important in cellular simulation. Simulation languages are typically orientated toward models with diverse elements with irregular interconnection topologies; they are concerned with statistical measurements such as average waiting time, length of queues, etc. and usually ignore the problem of simultaneous change of state of independent elements of the model. All of these characteristics make available languages awkward tools at best for performing cellular simulations.

New language constructs and the attendant checking and enhanced experimental useability are important to successful exploitation of cellular space models. We have given a formulation of cellular spaces that is broad enough to cover a variety of applications and rich in structure not provided by existing general purpose languages. In Chapter 3 a new language encompassing our formulation is developed which will substantially aid in conducting cellular simulations. It should be understood that while the author has implemented this language, it is the suitability of the

language that is important, and not his particular implementation.[1] Most important is that the language, however implemented, be suited to the task of simulating cellular spaces.

2.10 Formalization

This section concludes by formalizing the structure of the class of models realizable in our system. This characterization is based on the simulation system actually implemented in order to more concisely describe that system. It is presented here because the notation will be useful later in describing some of the details of the actual system.

Let $\mathbb{Z}$ be the set of integers both positive and negative; then $\mathbb{Z} \times \mathbb{Z}$ is an infinite two dimensional cartesian plane which we call P. Sp is a cellular space iff

$$Sp \subset P \cup E$$

where card (E) = 1 and $E \cap P = \phi$, and $Sp \cap P$ is circumscribable by a convex polygon.

Edge effects are defined by a function

$$e: P \rightarrow Sp$$

such that

$$\forall \alpha [\alpha \in Sp \Rightarrow e(\alpha) = \alpha].$$

The function e provides the mechanism for "wrapping around" to form cylinders, etc., as well as supplying a "default" state for neighbors outside of Sp; the constraint preserves the identity of cells within Sp.

---

[1] Implementation of such a new simulation language may be possible in one of the emerging "extendable" languages, such as MAD/I. Unfortunately none was available in a form suitable for our use. We have worked entirely in assembly language for the IBM 1800.

The general neighborhood N, i.e., common to all cells of the space, is an ordered set of elements of P:

$$N = (\Delta_1, \Delta_2, \ldots, \Delta_n) \text{ where each } \Delta_i \in P$$

The Δ_i are displacements from the central cell.

The state S of a cell consists of three parts:

$$S = X \times N' \times M$$

where

X is normally interpreted as the cell state

N' is the local neighborhood, and

M is an input selector.

While X is the part that is usually considered the "state" of a cell, the formal inclusion of N' and M will be exploited to provide dynamically changing neighborhoods and inputs. In particular N' may be functionally dependent on the time step and location while N is independent of both.

The total neighborhood of a cell $\alpha \in Sp$ is simply $\bar{N}(\alpha,t) = N + N'(\alpha,t)$, where "+" here means the ordered concatenation of ordered sets.

Input is defined as a function of an index, or selector, and time:

$$I: \{1, \ldots, m\} \times T \to Q$$

Each $I(m,\cdot)$ represents an input stream, and which input stream corresponds to a cell is determined by a substate of the cell data structure. The input to cell α is thus given by

$$I(M(\alpha,t), t).$$

The transition function f,

$$f: S \times S \times \ldots \times S \times Q \to S$$

$$S(\alpha,t+1) = f(s(e(\alpha+\Delta_1), t), \ldots, S(e(\alpha+\Delta_{n+n'}), t), I(M(\alpha,t), t)$$

Note that edge effects are incorporated within this formulation.

Output functions O_i map from the state space to a finite number of displayable graphics:

$$O_i: S \rightarrow \{G_1, \ldots, G_n\}, n \leq 127.$$

3. A LANGUAGE FOR CELL SPACE SIMULATION

The Cellular Space Simulation Language is a procedure oriented language designed to facilitate the writing and simulation of cellular space models. It provides a full complement of arithmetic and logical operators; a facility for defining structured data types and operators on those data types; and various data types, operators, and statements intrinsic to the simulation environment in which the translated object program will be executed.

The grammar for the language is syntactically an operator grammar. The lexical format is free form with respect to the input medium. The basic logical unit is the construction (terminated by a semicolon) in order to facilitate error recovery insofar as possible. Statements are one or more concatenated constructions. A sequence of statements is a program.

In describing the language we shall use the following conventions and notations. Example sections of source coding will be on separate lines with double indenting to keep them distinguished from the describing text. Source words are always in capital letters and this is also a useful cue. Syntactic descriptions of the language grammar will use a variation of the more common BNF notation.

Syntax will be described in its production (rather than reduction) form, e.g.,

<A> → B <C>

which may be read "the non-terminal symbol <A> may be replaced by the

symbol B followed by the non-terminal <C>". Tall square brackets will be used to designate several mutually exclusive possibilities, of which one must be present. Thus,

$$\langle A \rangle \rightarrow \left[\begin{array}{l} X\ \langle L \rangle \\ B \end{array}\right] \langle C \rangle$$

may be considered a shorthand for the two productions:

$$\langle A \rangle \rightarrow X\ \langle L \rangle\ \langle C \rangle$$

$$\langle A \rangle \rightarrow B\ \langle C \rangle$$

The latter will often be simplified by not repeating the left side, as in:

$$\langle A \rangle \rightarrow X\ \langle L \rangle\ \langle C \rangle$$

$$\rightarrow B\ \langle C \rangle$$

Curly brackets will be used to indicate a sequence that may be repeated an arbitrary (possibly null) number of times. If sub- and superscripts follow the right bracket, they are interpreted as minimum and maximum number of repetitions, respectively. Thus,

$$\langle A \rangle \rightarrow X\ \{Y\}$$

describes the same collection of terminal strings as

$$\langle A \rangle \rightarrow \langle A \rangle\ Y$$

$$\rightarrow X$$

and

$$\langle A \rangle \rightarrow \{X\}_1^3\ Z$$

is shorthand for

$$\langle A \rangle \rightarrow XZ$$

$$\rightarrow XXZ$$

$$\rightarrow XXXZ$$

We will not be concerned here with the subtleties of the differing parsing trees that might result from alternate interpretations of these shorthands. We distinguish between a description grammar which is used to convey the language to users, and an implementation grammar which is used explicitly for syntax directed parsing. Since we are primarily concerned with describing a language, the above conventions are both a convenience and in some cases more intuitively meaningful than a comparable BNF expression.

We admit that this dichotomy of grammars leaves ample opportunity for conflict and inconsistency between the description and the implementation. But since the compiler currently implemented used syntax directed methods only at the level of expressions and assignment statements, there is no formal implementation grammar for many aspects of the system. Accordingly, we will not be too embarrassed to occasionally use a suggestive non-terminal symbol such as <integer constant> without anywhere giving a syntactic definition of the symbol. The intention will be clear, and will accurately convey much semantic information about what is actually required by the compiler.

Summaries of the syntax, keyword tables, etc., may be found in Appendix A. Implementation of the compiler is discussed in Appendix B.

An example of a transition function written in this language is exhibited and discussed at the end of this Chapter. The reader may find it helpful to refer to Figure 3.3 for examples while reading the Chapter.

3.1 Procedural Aspects

3.1.1 Lexical Format

The lexical unit of the language is the atom. An *atom* is defined as

1) any of the non-alphanumeric characters except spaces or

quote, e.g. + - / %,

2) any string of alphanumeric characters delimited by non-alphanumeric or space or quote (period is considered a numeric and dollar sign an alphabetic),

3) any string of characters enclosed in quotes, (a quote character may be included within a quote string by two quotes in succession).

Note that space itself is never an atom and may always, and must sometimes, be used as a delimiter between atoms. For example, 123 is one atom while 1 2 3 is three atoms.

Lexically a *construction* is any sequence of atoms (except semicolon) which is followed by a semicolon. Note that the semicolon is considered part of the construction. Thus, the following construction consists of 14 atoms:

```
AX = A(5) + B[1, P$T];
```

A *statement* consists of a given number of constructions concatenated together and satisfying certain constraints. The statements of the language will be developed in detail shortly.

A *numeric atom* is an alphanumeric atom consisting of only numeric characters and at most one period. An *alphabetic atom* is an atom of only alphabetic characters. A λ-*atom* is an alphanumeric atom which is not a numeric atom and not a reserved atom (i.e., not used as a keyword or for any other predefined purpose). λ-atoms may be used as variable names or labels, defined as operators, etc.

If the first atom of a construction is a λ-atom and the second a colon then the λ-atom is implicitly defined as a constant of type LABEL. (See 3.1.2).

In general, the language allows only one kind of use for an atom. For example, no atom could be both a defined data type name and a label even though the correct use could almost certainly be inferred from the context. However, the dual use of "-" (minus sign) as both a unary and binary operator is so pervasive that it has been explicitly accommodated.

3.1.2 Primitive Data Types

The primitive data types of the language are:

1) INTEGER
2) REAL
3) BOOLEAN
4) LABEL
5) TEXT
6) FUNCTION

Each of the above atoms is a keyword whose use in the language is reserved for this particular purpose.

In general, the variables of the language may be of any of the above types. Constants of the above types will be recognized by their lexical properties as follows:

1) An INTEGER constant is any numeric atom without a period.
2) A REAL constant is any numeric atom containing exactly one period.
3) BOOLEAN constants are the atoms TRUE and FALSE.
4) LABEL constants are any λ-atom that occurs in the label field of a statement.

5) TEXT constants are recognized by their containing quotation characters.

6) FUNCTION name constants must be explicitly declared as explained later.

3.1.3 Declarations

There are two general declaration statements: DECLARE... and DEFINE... The first simply assigns a given attribute to each of a series of λ-atoms. The form of the statement is:

DECLARE <type>: <λ-atom list>;

For example:

DECLARE REAL: A, B, C;

DECLARE INTEGER: X, Y, GEORGE;

would establish A, B, and C as real valued variables and X, Y, and GEORGE as integer valued variables.

Acceptable λ-atoms for the <type> are any of the primitive data type names or any of the defined data type names (as explained below).

Two types of composition operations are available to generate data structures more complex than the primitives. The simpler of these is the fixed length array. The form is:

DEFINE <λ-atom> ARRAY <type> SIZE <integer constant>;

The interpretation is that the first <λ-atom> is defined as a <type> name which identifies a data structure consisting of a fixed number, given by the <integer constant>, of elements all of which are of type given by the <type>. Thus, to declare A an array of five reals and B a square array of seven by seven integers one writes:

```
DEFINE REAL5 ARRAY REAL SIZE 5;
DECLARE REAL5 : A;
DEFINE INT7 ARRAY INTEGER SIZE 7;
DEFINE SQINT7 ARRAY INT7 SIZE 7;
DECLARE SQINT7 : B;
```

The second composition operation provides for the definition of a *block* of contiguous data whose elements may be of diverse types. Blocks have also been called component structures or structured variables. The form of the statement is:

```
DECLARE <λ-atom>  BLOCK  [<type list>];
```

For example:

```
DEFINE QQSV BLOCK [REAL, INTEGER, INTEGER];
```

specifies that QQSV is a type name referring to a block consisting of a real number followed by two integers.

Either operation may be composed either with itself or with the other, thereby allowing complex data structures to be constructed in hierarchical fashion. By convention, the same structure may *not* be given more than one name.

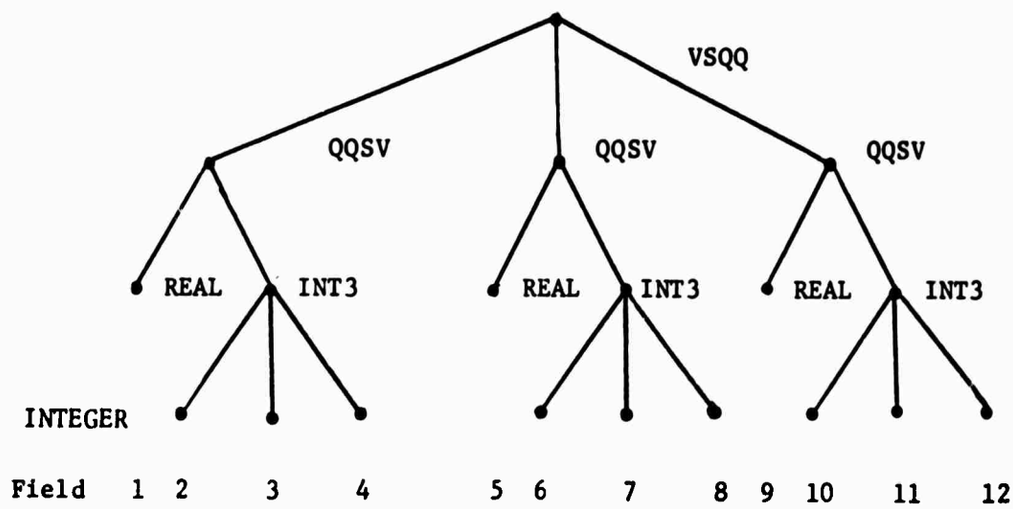
Components of a complex data type may be identified by a subscript following the data name. Subscripts are interpreted from left to right as identifying a lower data type in the hierarchical description of the data structure. Figure 3.1a presents a somewhat involved example.

In order to conveniently refer to primitive elements of a cell's data structure, we shall sometimes speak of an ordering of the fields of a data structure. A *field* is a substructure which has a primitive type. Field f_1 *precedes* field f_2 if in the subscript notation for referring to the respective fields, the subscript designation f_1 lexigraphically precedes

```
DEFINE  INT3  ARRAY  INTEGER  SIZE 3;
DEFINE  QQSV  BLOCK  [REAL,   INT3];
DEFINE  VSQQ  ARRAY  QQSV     SIZE 3;
DECLARE  VSQQ:  ABC;
```

ABC	is of type VSQQ
ABC (2)	QQSV
ABC (2,1)	REAL
ABC (1,2)	INT3
ABC (3,2,1)	INTEGER
ABC (1,2,7)	undefined

3.1a



3.1b

Example Data Structure Definition and Related Notation

Figure 3.1

the subscript designating f_2. In Figure 3.1a, ABC (1,1) precedes ABC (1,2,3) which precedes ABC (2,1), etc. Since this clearly gives a linear ordering, we shall speak of the first field, second field, etc. (One may equivalently think of drawing the "structure tree" of a data type, and then numbering the endpoints from left to right as illustrated in Figure 3.1b).

Either unary or binary operators may be defined on any available types. If a previous operator definition exists, it is replaced. A precedence is required for binary operators to place each new operator in appropriate relationship to other operators.

An operator definition consists of 1) a header statement, 2) a body consisting of a sequence of statements written in the language and 3) the "ENDOPR;" statement.

The forms of the header statement are:

DEFINE <λ-atom> UNARY <type>, <type>;

DEFINE <λ-atom> BINARY <integer>, <type>, <type>, <type>;

Each of the <type> names must be previously defined. The <integer> gives the precedence and the last <type> gives the type of the result. The other <type> (s) specify the required arguments.

The body of the definition may be empty. If it is, then a subroutine CALL to the operator named will be generated with two or three arguments which are addresses of the appropriate data. If the body is non-empty, then it will be assumed to define a routine to compute the needed result. Arguments may be referred to by the form % <integer> where the ordinal value of the integer constant specifies the argument.

The last statement delimits the scope of the operator definition.

As an example, suppose that variables of type REAL3 are being used to represent vectors in a three dimensional (physical) space. An operator for evaluating the dot product of two vectors might be defined as follows:[1,2]

```
DEFINE REAL3 ARRAY REAL SIZE 3;
DEFINE $DOT$ BINARY 15, REAL3, REAL3, REAL;
%3 = %1(1)*%2(1) + %1(2)*%2(2) + %1(3)*%2(3);
ENDOPR;
```

The PARAMETER declaration is designed to allow mnemonic names to be used in place of integer constants (primarily when used as subscripts.) This declaration is recommended to resolve the awkward choice resulting from block data types. A numeric subscript is non-intuitive but gives a known compile time data type, while a heuristically choosen and suitable valued variable does not permit a known compile time data type. Use of a parameter removes the problem.

The form of the statement is

PARAMETER { ($\lambda\{,\}_0^1$ <integer constant>)};

It consists of a series of ordered pairs. The first of the pair is replaced by the second whereever encountered. For example the following

---

1 Note the quite different meanings of the words "dimension" and "vector" when used in their physical and computer science senses. In this example, a three dimensional vector in physical space is represented by a one dimensional array (vector) or size (length) three.

2 Although not necessary, we find it convenient to use λ-atoms whose initial and final characters are dollar sign as binary operators, and λ-atoms with final (and not initial) dollar sign as unary operators. This practice is followed through out this thesis.

```
PARAMETER  (XYZ 12)  (ALPHA, 0);
X = ABC(XYZ, ALPHA);
```

is equivalent to

```
X = ABC(12,0);
```

Note that the parameter substitution becomes effective at the point of definition and is not retroactive to previous statements. Also, note that parameter atoms may not be "chained". Further, the substitution is actually performed before syntactic parsing and hence, the integer constant is actually used by the parser. This permits the type result of a subscript of a BLOCK structure to be known at compile time.

3.1.4 Executable Statements

The basic executable statements of the language are an assignment statement, an unconditional branch statement, conditional branch statements, and an iteration statement. Several miscellaneous statements are also available.

3.1.4.1 Assignment

The most basic statement of the language is the assignment statement. The most succinct way to present the acceptable forms of this statement is via the productions of a grammar. Such a description is found in Figure 3.2.

The following observations are made about this description:

<assignment> → <left des> = <exp>

<left des> → λ

→ λ (<exp list>)

<exp list> → <exp>

→ <exp list>, <exp>

<exp> → <exp> θ <exp>

→ φ <des>

→ <des>

<des> → <left des>

→ (<exp>)

→ λ! (<exp list>)

→ λ!

→ <lsv>

<des> → (<assignment>)

<lsv> → [<exp list>]

Syntax of Assignment Statement
Figure 3.2

1) The symbols λ, θ, and φ are not particular terminal symbols but designators for the class of λ-atoms, binary operators and unary operators respectively. The lexical parsing actually performs the necessary assignment of an atom to these classes, if appropriate, prior to parsing.

2) An explicit "operator", i.e. the exclamation mark, is used to designate function calls. This convention is similar to that of MAD [12]; however a different atom has been chosen to avoid another usage for the period (which is used in MAD).

3) In this descriptive grammar the production

<exp> → <exp> θ <exp>

obviously introduces an ambiguity into the resulting language. In contrast to typical applications, the grammar rules are not used here to establish the relative precedence of a multitude of binary operators. In contexts where an ambiguity exists in parsing an input string, as for example in

A + B * C

the order of association is resolved by an extra-grammatical attribute of all binary operators, its (single) precedence value. The resulting organization has the virtue of making the grammar invariant with respect to the number and relative precedence of binary operators, and accordingly new binary operators can be introduced with absolutely no impact on the syntactic parser.

A special compile time binary operator, atom "@", is used for various purposes. The operator accepts as a left operand an argument of any type and as right operand only an argument of type TEXT. The intention is to allow *ad hoc* extentions to the compiler with a minimum of effort. Three

immediate applications are the following:

Where BLOCK data structures are used with variable subscripts, in general the <type> of an expression is not determinable.

For example:

```
DEFINE ABLE BLOCK [INTEGER, REAL]; DECLARE ABLE: MARY;
X = MARY (I);
```

MARY (I) will be of type INTEGER if I has value 1, and type REAL if I has value 2. However, the compiler does not support dynamic data types at run time, and the type of every (sub-) expression must be known at compile time. Thus, the above assignment could not be compiled as it stands. By using the @ operator followed by a <type> atom, such situations may be resolved.

Thus,

```
X = MARY (I) @ "INTEGER";
```

would compile and treat MARY (I) as an integer regardless of the value of I. The operator may also be similarly used to override a known <type> in favor of a different one.

At some point it may be desirable to introduce new primitives which have lexically the same constant forms as existing ones. 0.5 is the most natural lexical representation whether the atom is REAL or FIXED-POINT. But

```
0.5 @ "REAL"
0.5 @ "FIXEDPOINT"
```

clearly indicates which conversion is desired.

Real constants in the commonly available scientific (or E-) format can be conveniently introduced by using the exponent as the second argument thus:

0.5 @ "E25"

1.873 @ "E-6"

3.1.4.2 Unconditional Branch

The form of the unconditional transfer is

GOTO <exp>;

where <exp> is of type LABEL. Control passes to the statement with the designated label.

3.1.4.3 Conditional Branch

The conditional branch statements are of the following forms:

IF <exp>;

ORIF <exp>;

ELSE;

ENDIF;

These are interpreted the same as the compound conditional in the 7090 MAD language [12]. In particular, note that exactly one "ENDIF;" must follow the "IF..." to determine the scope of the sequence. The "ORIF..." may be used any number of times, and the "ELSE;" at most once. These statements delimit mutually exclusive sequences of statements of which the first true condition will enable its corresponding body to be executed.

If IF, ORIF, ELSE, and ENDIF represent their respective statements and α any valid sequence of statements, then the following defines legal uses of the conditional branch:

<legal IF> → IF α{ORIF α} {ELSE α}$_0^1$ ENDIF

3.1.4.4 Loop Statement

The loop statement is of the form:

```
LABEL1:  LOOP <assignment>; <exp>; <exp>;
         ...
LABEL2:  ENDLOOP;
LABEL3:  ...
```

The interpretation of this statement is similar to that of 7090 MAD. The left side of the assignment specifies the controlled variable for the loop. The assignment is performed, then the second expression is evaluated. If TRUE, the loop is terminated; otherwise, the loop body is executed. The loop body may not be executed at all. The ENDLOOP statement returns control to the loop header statement where the controlled variable is incremented by the first expression and the termination test performed again. For example:

```
         I = 5;
         LOOP X(I) = DATA(I); DATA(I); X(I)$GT$ 0;
         X(I) = 0;
         ENDLOOP;
```

is equivalent to

```
         I = 5;
         X(I) = DATA(I)
         GOTO T1;
T3:      X(5) = X(5) + DATA(I);
T1:      IF X(I)$GT$ 0;
         GOTO T2;
```

```
            ENDIF;
            X(I) = 0;
            GOTO T3;
T2:         CONTINUE;
```

3.1.4.5 Miscellaneous Executable Statements

The miscellaneous statements include:

1) CONTINUE;

This serves as a convenient way to introduce a label.

2) EXECUTE <exp>;

This allows direct subroutine calls.

3) PRINT <exp. list>;

READ <λ-atom list>;

These are not properly part of the simulation language. They are included for diagnostic and developmental purposes and may be a suitable nucleus for I/O statements of a "free standing" language. They are also a convenience in developing and debugging transition functions.

4) RETURN;

This statement terminates the transition function and returns control to the run time system.

5) ENDPROG;

This indicates the physical end of the program.

If program execution "flows into" this statement it will be functionally equivalent to a RETURN.

3.1.5 Literal Structured Variables

A *literal structured variable* (abbreviated LSV) is a non-primitive variable or constant whose data structure is explicitly represented in its lexical representation in a rather natural manner. For example, if data type ALPHA is defined by

```
DEFINE ALPHA BLOCK [REAL, INTEGER, INTEGER];
```

then [1., 2, 3,] is an instance of a constant of type ALPHA. Further if:

```
DEFINE BETA ARRAY REAL (2);
DEFINE GAMMA BLOCK [INTEGER, BETA];
DECLARE BETA: ABC;
DECLARE GAMMA: DEF;
DECLARE REAL: X;
```

then [1, [2., 3.]] and [1, ABC] are LSVs of type GAMMA.

Literal structured variables may be used anywhere to the right of an assignment operator where a variable of the same type would be used. To continue the last example, if = (assignment) is given its usual interpretation, then the statement

```
DEF = [1, [5., 7.]];
```

would be functionally equivalent to

```
DEF[1] = 1;
DEF[2,1] = 5.;
DEF[2,2] = 7.;
```

and

```
DEF = [2, ABC]
```

would be equivalent to

```
DEF [1] = 2;
DEF [1,1] = ABC [1];
DEF [2,2] = ABC [2];
```

More generally

```
DEF = [1, ABC + [X, 0.5]];
```

corresponds to

```
DEF(1) = 1;
DEF(2,1) = ABC (1) + X;
DEF(2,2) = ABC (2) + 0.5;
```

The above examples illustrate the meaning of operations involving LSVs and do not describe the order in which the computations are performed. See Appendix B for a discussion of the implementation of LSV's.[1]

An LSV has a <type> which may be inferred from its lexical form. Thus, in the above examples, the occurrence of [1, [2., 3.]] is sufficient to recognize that its type is GAMMA. Accordingly, there must be a declaration for each type of LSV that may occur.

Subscripts may not be used with LSV's, e.g.,

[1, [2., 3.]] (2)

is not an acceptable alternative for

[2., 3.]

While such an interpretation is quite natural in the context of the present development, it does not add anything to the language, and indeed, detracts by allowing the user to obscure what computation is being performed.

---

[1] We note that structured variables are not manipulated via pointers, i.e., the whole data block is physically moved during an assignment operation. However, there are certain approaches that can be used to minimize physical data movement with an LSV.

3.2 Simulation Oriented Aspects

Certain data type names, operators and statements are included in the language explicitly for the simulation model. These will be described in three groups: data structures, entry points, and operators.

3.2.1 Data Structures

3.2.1.1 Cell Data Structure

The λ-atom CELL is reserved as a <type> naming the data structure for the cells of a simulation. The user must include a "DEFINE CELL..." statement that specifies this information. The global variable NEWSTATE is implicitly declared to be of type CELL. The value of NEWSTATE becomes the value of the current cell on the next time step. The value NEWSTATE on entry to the transition function is the current state of the current cell. Thus substructures not assigned new values by the transition function, if any, will remain unchanged.

The preferred way to refer to cells is as elements of the array variable CELLS. Thus, CELLS (1) is the first neighbor of the current cell, CELLS (2) is the second,[1] etc. Further subscripting may be used to refer to substructures of the cell data state. One may imagine that the following declarations have been made:

```
DEFINE CELLARRAY ARRAY CELL SIZE ?;
DECLARE CELLARRAY: CELLS;
```

where the size of the array may not be known at compile time.

---

[1] One may of course assign a value to a variable, e.g., LN, and then write CELLS(LN) to refer to the "left neighbor" if such mnemonics are convenient. Chapter Five comments further on this naming problem.

The declaration

DIMENSION <integer>;[1]

is used to implicitly define the attribute type COORD as

DEFINE COORD ARRAY INTEGER SIZE <integer>;

Variables of type COORD are interpreted as relative vectors in the coordinate space of the cell space.

Moreover, it is permissible for the definition of type CELL to include one or more components of type COORD. This allows the cell state itself to specify some (or all) of its neighbor cells. Since this specification is used at run time and is part of the cell state specification, it follows that a dynamically changing neighborhood can be modeled. The neighbors specified in the current cell by components of type COORD are automatically accessed and made available as neighbors through subscripts of variable CELLS. No explicit attention need be given to the difference between the general neighborhood and the local neighborhood, except that the local neighborhood can be changed for successive time steps by the transition function.

A variable called INPUT may be accessed to obtain the input to the current cell provided:

1) A component of attribute type SELECTINPUT is included in the definition of attribute type CELL, and

2) An attribute type is DECLARED for the variable.[2]

The linkages to input routines are specified at the time of loading the simulation system. Note that the component of type SELECTINPUT is

---

[1] In the current implementation, the DIMENSION is fixed at 2 and may not be changed.

[2] In the present implementation, the attribute type is predefined as INTEGER and may not be changed.

an integer whose value may be changed at run time.

3.2.1.2 External and Initial Cell States.

The cell state to be used for the external cell is declared via:

DECLARE EXTERNAL: <value>;

The value must be a variable or constant which is of data type CELL. It may be an LSV provided no operators are present.

Similarly, the initial state of the space may be declared by:

DECLARE INITIAL: <value>;

where the <value> is as above.

Declaring either of the external or initial states is optional. If no initial state is given, it is the users responsibility to establish his desired starting state.[1] If the EXTERNAL state is not declared and an external cell state is required by the simulator, a vector of all zeros will be used. (See the discussion under Default Specifications, 3.2.4.)

3.2.1.3 Neighborhood, Size of Space and Edge Declarations.

The declaration

DEFINENBHD $\{\text{<coordinate>}\}_0^{max}$;

is used to define the general neighborhood relation, common to all cells of the space. <coordinate> is a constant LSV of type COORD. Neighbor cells correspond with subscripts of variable CELLS in the order given in the neighborhood definition. The five neighbors so commonly used might be defined thus:

DEFINENBHD [0,0] [1,0] [0,-1] [-1,0] [0,1];

---

[1] This might be done with the U command or from a previously saved state, as discussed later.

If data type CELL contains elements of type COORD, these will correspond with successive subscripts of CELLS in the order[1] in which they occur in the specification of type CELL (i.e., following the numbers for general neighborhood).

To specify the size of a simulation, use

DEFINESIZE {<coordinate>}$_3^{max}$;

where <coordinate> is an LSV of type COORD. For example

DEFINESIZE [-5,-5] [-5,5] [5,5] [5,-5]

defines a square array of eleven by eleven cells. At least three coordinates are required to define a polygon. Coordinates are assumed to be given in clockwise rotation around the figure.

The action of the simulator when accessing states for cells outside the space boundaries (as discussed in 2.3) may be one of several standard options, or the experimenter may elect to specify his own.
The declaration is

DEFINEEDGE [<label> | XWRAP | YWRAP | TORUS] ;

If a <label> is given, the <label> is defined as an entry point to a procedure to perform the desired action. At entry the global variable LOCATE of type COORD contains the coordinates needing modification. The edge routine should modify LOCATE to lie within the cell space and RETURN. Alternatively, if a RETURN is executed with LOCATE unchanged, then the

---

[1] The ordering used is the lexigraphic order defined earlier.

system will attempt to use an EXTERNAL cell value.

The declaration is optional. If not used, any reference outside the space will access the EXTERNAL cell value.

Intuitively the XWRAP option causes the right edge (increasing x direction) to be "wrapped around" and made adjacent to the left edge (decreasing x direction). Since the shape of the space may be any convex polygon, rows of constant Y may be of differing length. Thus each row will be wrapped independently of the others. YWRAP is similar. The TORUS option wraps in both directions simultaneously. (Care must be taken to handle cells that exceed the X and Y limits simultaneously in order to be able to define the wrapping). Note that if the space is rectangular, these options will give the usual respective cylindrical or toroidal geometries in a Cartesian space. If the shape is not rectangular, these transformations are at least well defined (and, I venture, at least as reasonable a notion of wrap around as can be formulated).

The actual transformations are tedious to define. They are given in Appendix C (Section C.2.1).

3.2.2 Entry Points

In addition to the transition function itself it is desirable to include in the same body of code several entry points to precedures for:

1) Providing certain commands intrinsic to the particular model being simulated.

2) Providing the needed MAP functions for displaying cell states.

The name of the transition function itself is declared via:

DECLARE <λ-atom> NAME;

the <λ-atom> must also satisfy the naming conventions of the operating

system and is intended to identify the object code to the operating system.

Experience has shown the utility of two additional entry points, to perform certain operations before and after a full transition of the cell space has been computed. Accordingly the declarations

DECLARE <λ-atom> PRETRANSENTRY;

DECLARE <λ-atom> POSTTRANSENTRY;

are provided. These enable the transition routine to accumulate certain kinds of statistics or perform other kinds of "housekeeping" services for the transition function.

Entry points to mapping functions are declared thus:

DECLARE <λ-atom> MAPENTRY <integer>;

The <λ-atom> is a label in the program. The <integer> must lie in the range zero to nine and associates an external number to be used to select that map.

The mapping function has one implicit parameter, the variable NEWCELL. The result of the map must be an integer in the range 0 to 126 which identifies the graphic image that will be used to display the cell state. The result is made known to the system by assigning the value to the system defined INTEGER variable GRAPHIC.

Typically one map will be desired for each field of a cell's state, but any others may be used as desired.

The "user" entry point is declared thus:

DECLARE <λ-atom> USERENTRY;

The λ-atom, a LABEL, so declared is the entry to whatever personal commands the experimenter wishes to define with his particular simulation. This entry point is entered as a result of typing the "U" command on the keyboard

followed by a text string as explained in the next chapter. This string is collected and made available to the USER entry in the system defined integer array USERINPUT. USERINPUT (1) is the length of the array for this call. The array will contain one integer (the character code) for each character and a three entry sequence for each converted data constant. The form of a converted data constant is 1) the character for "=" (equals), 2) one of the characters "B", "I", or "R" for BOOLEAN, INTEGER, or REAL designating the type of the constant and 3) the value itself.[1] For example, the typed input string

"A1B=I-70 =R8.2=BTRUE"

results in the USERINPUT array containing:

USERINPUT (1)	=	14
	=	code for letter A
	=	code for letter 1
	=	code for letter B
	=	code for letter =
	=	code for letter I
	=	integer value -70
	=	code for letter space
	=	code for letter =
	=	code for letter R
	=	real value 8.2
	=	code for letter =
	=	code for letter B
USERINPUT (14)	=	boolean value TRUE

This allows a reasonably complex command interpreter to be written quite easily.

To allow for command checking and reporting of ill-formed user commands, the USER routine must set the system defined BOOLEAN variable USEROKAY either TRUE or FALSE to indicate the command was accepted or rejected,

---

[1] On machines where the representation of real values requires more memory (words, bytes) than integers, this (unfortunately) must be explicitly recognized by the USER routine in scanning the USERINPUT array.

respectively. A rejected command will be reported to the experimenter via the console keyboard-printer.

3.2.3 Operators

The unary operator UNDEF$ requires an operand of type TEXT. It is basically a call to the run time system declaring that (for what ever reason) the transition function for the current cell is undefined and a value can not be returned. A later section will discuss the effect this has on the system and actions open to the user for handling this situation.

3.2.4 Default Specifications

Careful attention has been given to designing the system with a natural set of default specifications for use where some declarations are not given by the user. The following default conventions have been adopted:

SELECTINPUT need not be used. If not used, no input is possible.

EXTERNAL and INITIAL state values have a default specification consisting of an appropriate number of (machine) zeros to "fill-up" a constant of type CELL.[1]

Pre- and Post-transition entries need not be declared.

DEFINENBHD will default to the standard five neighbor neighborhood:

[0,0] [1,0] [0,-1] [-1,0] [0,1].

DEFINESIZE will default to the largest square array that can be accommodated by the implementation.[2]

---

[1] On the current computer, hardware zeros will interpret as a BOOLEAN FALSE, an INTEGER "0", and a REAL "0.0".

[2] Currently 32 by 32 as follows: [1,1] [1,32] [32,32] [32,1].

DEFINEEDGE will default to using the EXTERNAL state value.

A default USER routine that will always respond with an error to any input is provided. When RETURNing from a USER routine it is only necessary to explicitly set the value of USEROKAY if it is to be TRUE.

All maps default to returning a value of zero.

3.3 An Example: MOD8

As a simple example we shall consider a cellular space in which the computational aspects of the transition function are trivial. The transition function simply computes the sum modulo 8 of the neighboring cells. We shall use the common five cell neighborhood. With appropriate choice of initial states such a cell space will exhibit a behavior that is esthetically quite pleasing to observe.

In order to make the example a bit more interesting, the state of each cell is defined to consist of a SELECTINPUT field as well as the INTEGER field that represents the "logical state". The transition function tests the SELECTINPUT field and if non-zero assigns the next cell state from INPUT rather than doing the normal computation. (One can imagine this as providing a "forcing function" in the cell space.)

The transition function and cell space specification is shown in Figure 3.3 to which the following annotations are offered:[1]

Lines 1-5 simply declare the various entry points. Lines 6-8 define the CELL data structure and declare some variables. Line 9 declares what state is to be used for external cells. Lines 10-11 define the size and

---

[1] Lines in the program listing are referred to sequentially from the beginning of the program or relative to a program label. Comments and blank lines are not included in this count.

```
*THIS IS THE NAME OF THE TRANSITION ROUTINE ;
          DECLARE MODE NAME;
*THESE DECLARE THE USER AND 3 MAP ENTRIES;
          DECLARE USERR USERENTRY ;
          DECLARE MAP1 MAPENTRY 1;
          DECLARE MAP2 MAPENTRY 2;
          DECLARE MAP3 MAPENTRY 3;
          DECLARE INTEGER: T,N;
*A CELL CONSISTS OF TWO INTEGERS, THE FIRST OF WHICH
 IS THE INPUT SELECTOR;
          DEFINE STATE BLOCK < SELECTINPUT, INTEGER >;
          DECLARE STATE: CELL;
*THE TYPE OF THE EXTERNAL CELL STATE IS IMPLICITLY 'CELL';
          DECLARE EXTERNAL:EXT;
          DEFINENBHD     <0,0> <0,1> <1,0> <-1,0> <0,-1>;
          DEFINESIZE <1,1> <1,20> <20,20> <20,1>;

MODE:     IF CELLS(1,1) $NE$ 0;
*'INPUT' IS IMPLICITLY OF TYPE INTEGER;
*'NEWSTATE' IS ALSO IMPLICITLY OF TYPE SAME AS 'CELL';
               NEWSTATE(2) = INPUT;
          ELSE;
               T=0;
*'NUMNEIGH' - THE NUMBER OF NEIGHBORS - IS A GLOBAL VARIABLE
 OF TYPE 'INTEGER';
               LOOP N=1; 1; N $GT$ NUMNEIGH;
                    T = T + CELLS(N,2);
               ENDLOOP;
               NEWSTATE(2) = T-(T/8)*8;
          ENDIF;          RETURN;

USERR:    IF "=" $EQ$ USERINPUT(2) $AND$ USERINPUT(3) $EQ$ "1";
               EXT (2)=USERINPUT(4);
               USEROKAY = TRUE;
          ELSE;
               USEROKAY = FALSE;
          ENDIF;  EXT(1) =0;  RETURN;

MAP1:     GRAPHIC = NEWSTATE(1); RETURN;

MAP2:     GRAPHIC = NEWSTATE(2); RETURN;

MAP3:     IF NEWSTATE(2) $GT$ 3;
                         GRAPHIC = 4;
          ELSE;
                         GRAPHIC = 0;
          ENDIF;
          RETURN;
          ENDPROG;
```

MODE CELL SPACE
FIGURE 3.3

neighborhood of the space.

Lines MOD8 to MOD8+8 give the transition function. If there is input to the cell (line MOD8), the input value becomes the cell state (line MOD8+1). Otherwise (line MOD8+2) the values of all neighbors are summed (lines MOD8+3 to MOD8+6) and the modulo 8 result becomes the cell state (MOD8+7). If the user is confident that the neighborhood declaration would always contain five neighbors then lines MOD8+3 to MOD8+6 could readily be replaced by the simpler:

```
T=CELLS(1,2)+CELLS(2,2)+CELLS(3,2)+CELLS(4,2)+CELLS(5,2);
```

Lines USERR to USERR+5 give a routine to change the value of the external cell state under keyboard control.

The remaining lines define map functions to display the SELECTINPUT field, the INTEGER field or a function of the INTEGER field.

Note that the value of the EXTERNAL variable EXT is undefined at compile time. The user is advised define it with his "U" command before any transitions are computed.

4. THE RUN TIME ENVIRONMENT

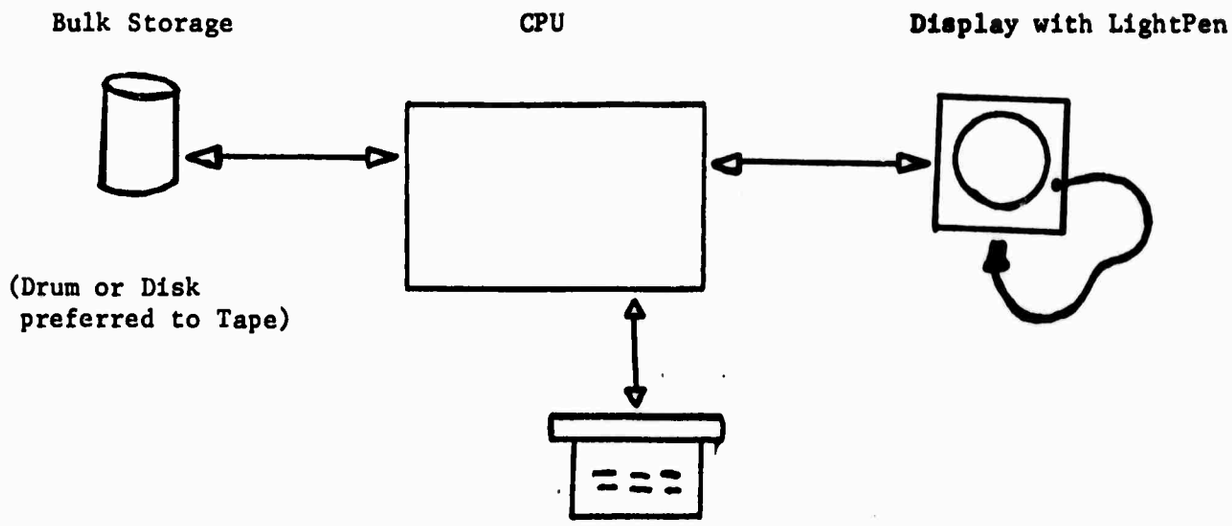
The Run Time Environment is the collection of routines and supporting software that is assumed to be available at the time that a simulation is taking place, exclusive of what is typically called system software. That is, the Run Time Environment is the application oriented subsystem, although details of its form will naturally be influenced by the form of the underlying system software.

The particular hardware on which this author worked consisted of two dissimilar small general purpose computers, one with a graphic display and the other with a disk storage unit. The two computers are interfaced to provide interactive processing and sharing of bulk storage and display facilities. For the purposes of this exposition the fact of two CPU's is not relevant. While the potential for parallel processing has been exploited to some extent, this is not crucial to the formulation of the system to be developed here. Accordingly the description of the Run Time Environment presented in this Section will not assume more than the conventional computer configuration shown in Figure 4.1. Further details of the two computer implementation may be found in Appendix C.

In preparing a simulation for the CSS there are a number of distinct specifications and programs that must be supplied. These are:

A. Specifications

   1. Cell State Data Structure
   2. Neighborhood Relationship



Conventional Computer Configuration
Figure 4.1

3. Size of the Cell Space

4. Initial State of the Cell Space

B. Programs

1. Transition Function

2. Mapping Function (s)

3. User Defined (personal) Commands

4. Input Function (s)

The cell state data structure (A.1) and the general neighborhood relationship (A.2) are specified as part of the transition function (B.1). The neighborhood may be redefined at run-time by keyboard commands provided the number of neighbors remains unchanged. The size of the cell space (A.3) is specified with the transition function (B.1) or commands that result in a dynamic assignment of core storage.

The initial state (A.4) of a cell space may be established in one of two ways:

1) Using the "DECLARE INITIAL:..." statement, or

2) Using the Restore command (discussed shortly) to establish the current state from a previously save state space.

In cases where it is not sufficient to use the "DECLARE..." statement to specify the initial state, it will be expedient to prepare a special transition function whose sole purpose is to assign the desired state to the space. This state can then be saved by the Keep command (discussed shortly) for later use.

Preparation of a transition function (B.1) is essentially an "off-line"

activity with respect to the simulation proper. A transition function is written in the language defined herein, then compiled, and loaded. Loading passes control to the Run Time Environment to accept the remaining specifications and user commands, and then start the simulation. Mapping function(s) (B.2) and user commands (B.5) are included with the transition function. Inputs (B.4) are defined via a keyboard command after the system is loaded.

The steps required to initiate a simulation will not be discussed here. These actions are heavily dependent on the operating system and its job control language. It is useful, for example, to be able to specify and re-specify transition function routines dynamically. However, where dynamic storage management and dynamic loading are not available and where core is too scarce to permit simultaneous loading of several alternative transition routines (as was the case for this author) certain sometimes awkward compromises are necessary. These considerations are on the fringes of the objectives of this thesis. The particular scheme used by the author is described in Appendix C. It is left to the reader to supply the simple on-line commands necessary for some of these "nice" features.

Once a simulation has been initiated a number of command facilities are available to provide the user with control over the course of the simulation. Most of the commands may be given either from the keyboard or via light buttons on a display menu. For various reasons, other of the commands may be given only from the keyboard or from the display.

4.1 Keyboard Command Facilities

A number of commands are provided at the system keyboard-printer (a 33KSR) for controlling the action of the simulator. Most commands are executed immediately as read in from the keyboard. Section 4.1.2 will discuss a simple facility for constructing micro-programs to be executed by the command interpreter.

A command consists of one, two or three contiguous letters (i.e., not separated by space, carriage return or other non-letter) or any other single character, (except space and carriage return which are ignored.) Commands may have parameters and the form of these is dependent on the particular command.

4.1.1 Immediate Commands

The most basic commands, all of which require no parameters, are tabulated in Figure 4.2. The brief explanation contained there should adequately explain the function of each, except for Back-up. Further commands are tabulated in Figure 4.3 and discussed here.

The Back-up command restores the state of the simulation to that of the previous time step. This is possible because of the duplicate data structure adopted as discussed in Chapter 2. In general, transition functions are not backwards deterministic (i.e., the state at time t can not be uniquely determined from the state at time t+1). Accordingly Back-up may only be performed if there is a valid preceding state data structure to which to refer. For example one may not Back-up from the initial state or immediately after a Restore operation.

Command	Mnemonic	Explanation
N	Next time step	Compute next time step and display under current map.
B	Back-up	Back-up to previous time step and display under current map. May be done only once at a given time.
S	Simulate	Compute successive time steps and display under current map until halted.
H	Halt	Halt simulation.
C	Clear and reset	Clear and reset simulation to its initial conditions.
X	Redraw	Re-compute image for this time step-presumably after changing the map.
0,1,...,9	Map Numbers	Select the indicated map number for subsequent images.
D	Debug	Call in and transfer to system debugging facilities.
PIC	Picture	Activate shutter control on movie camera to take a picture of the current cell space display.

Commands Without Parameters
Figure 4.2

Command	Mnemonic	Parameter	Explanation
K	Keep current space state	File number, 0 or N	Save current state of cell space in designated logical file
R	Restore	File number	Restore saved state of cell space to be current state.
U	User	Character String	Call USER entry point in transition function with character string as parameter. Provides extension of command language.
I	Input	Input Stream	Define cell inputs. See text for elaboration.
TTL	Title	Character String	Declare title to use on cell space display.

Commands with Parameters
Figure 4.3

The two commands K (Keep) and R (Restore) are concerned with check-pointing and recovery of a given cell space state. The Keep command must be followed by an integer parameter, then by one of the letters "O" or "N". The integer is interpreted as the name of a file in the logical file system where the state information is to be stored. The letter designates whether this is to be an old or new file. If old, the previous contents will be replaced; if new, a new file will be created and the data written into it. If old is specified and no file exists, the operation is aborted and an error message given.

The Restore command is simply the inverse, i.e., the current cell state is established from the external file.

The following data is preserved by the Keep operation in order to check for proper reloading:

1. Title and current time step
2. Number of words per cell
3. External cell state, if any
4. Description of cell space boundaries (internal data structure)
5. State of the cell space

During the Restore operation items 2 and 4 will be compared with the corresponding current simulation data, and if the same, the operation proceeds. If not the same, the user may elect to proceed anyway under the following assignment rules:

Let A be the cell space currently in core, with α a coordinate variable. Thus, $A(\alpha)$ is the state of the (single) cell at coordinates α. Similarly for B and β defined in the file.

1) If $\gamma \epsilon S_A$ and $\gamma \epsilon S_B$, then $A(\gamma) = B(\gamma)$.

2) If $\gamma \epsilon S_A$ and $\gamma \epsilon S_B$ and E is the external state for S_B, then $A(\gamma) = E$.

3) Otherwise $A(\gamma)$ = "undefined".

The intent is to allow a means, albeit less than ideal, to change the boundaries of a simulation in "mid-course" and proceed.

The command USER is intended to allow extensions to the command language to provide operations dependent on the particular model being simulated. A character string is collected with data constant substrings being converted to binary form, and the USER entry point of the transition function is called giving the string as parameter. This string may be interpreted in any desired manner to provide whatever special functions are needed. Exit from USER returns a value TRUE or FALSE indicating all okay or some error. If an error is indicated, it will be reported via the normal command language error messages.

Data constant substrings are indicated by a leading "=" (equal sign), followed by a letter indicating the desired type conversion, followed by the constant itself. The constant is converted according to the indicated type. For convenience, the "I" may be deleted for integer constants and the single letters "T" and "F" will serve as the boolean constants TRUE and FALSE respectively. In any case the USERINPUT array will have the "full" form as described in 3.2.2.

INPUT is defined from the keyboard via a command 'I' with a parameter string that defines a sequence of integer values.

The form of the command is:

<I command> → <integer1> {<repeat group>}$\begin{bmatrix} R \\ T \end{bmatrix}$

<repeat group> → (<integer2> <repeat group>)

→ {<integer>}

where

<integer1> specifies the input selector for this input stream.

R specifies that when the input has "finished", it will restart at its beginning. This is equivalent to an additional layer of parentheses with an infinite count.

T specifies that when the input has finished, it will not restart; it is a one time only input.

A <repeat group> is a string of integer values optionally surrounded by parentheses with a repetition count <integer2> following the left parenthesis. Nesting is permitted.

Using this command quite complex input streams over the set of integers may be defined. In fact, the set of input streams corresponds to the regular events over the alphabet consisting of the set of integers.[1]

4.1.2 Deferred Execution via Micro-Program

A group of commands is concerned with defining and "executing" a micro-program built up of other commands. There are tabulated in Figure 4.4.

The command SAV simply saves in an internal character buffer all following characters up to and including "#" (pound sign). Even carriage return is saved so that the length of a micro-program is not restricted to a single input line. The command GTM will cause control to pass to the micro-program which the command interpreter will process until one of several terminating conditions occurs. A micro-program may be invoked more than once.

---

[1] To verify this requires that the three operations used to describe regular events (concatenation, union, and star) have equivalents in this command description. Note that a regular expression describes a set of regular events whereas an input command describes exactly one input. Concatenation of symbols is available as such in defining inputs. The union operation is available as alternative input definitions. The set of input definitions which are the same except for the value of a given repetition count corresponds to the set given by the star operator, eg. $\{X^*\} = \{ (yX) \mid y \text{ is a non-negative integer}\}$.

Command	Mnemonic	Parameter	Explanation
SAV	Save micro-program	Char string	Save micro-program for deferred execution. Char. String is terminated by "#" character.
GTM	Go to micro-program	No	Transfer control to micro-program
(	Start repeat	Count	Start repeat loop in micro-program
)	End repeat	No	End of repeat loop
?	Terminate micro-program	No	Terminate micro-program if sense switch 17 is "on".

Commands for Deferred Execution
Figure 4.4

There are three commands that can be used within micro-programs that can not be used otherwise. The two commands "(" and ")" must occur in balanced pairs; nesting is allowed to a depth of 5. The pair of parentheses specify a sub-string that is to be repeated a given number of times. This repetition count must immediately follow the "(" and the remainder of the string is the repeated part. For example the following input

"X SAV (2 1X 2X (2N)) #GTM"

is equivalent to

"X SAV 1X 2X N N 1X 2X N N #GTM"

The command "?" will cause a micro-program to terminate if sense switch 17 of the computer is on.

A micro-program will terminate for any of the following reasons:

1) running off the end (a normal case)
2) executing "?" with switch 17 up (normal case)
3) executing SAV or GTM (errors)

4.1.3 Commands for "Undefined" transitions

An invocation of the UNDEF operator will cause the following sequence of events:

1) The current space transition computation is suspended.
2) A message announcing the occurrence and concluding with the TEXT operand is typed on the console printer.
3) If a micro-program is in process, it is suspended. Control returns to the keyboard.
4) The Cell Space Display is displayed with a "box" around the cell at the undefined transition position.

At this point the user has available the full resources of the simulation system for examining the conditions leading to the situation and correcting them as appropriate. In this "suspended" status the following commands may

not be used: Next, Simulate, Back-up and Restore.

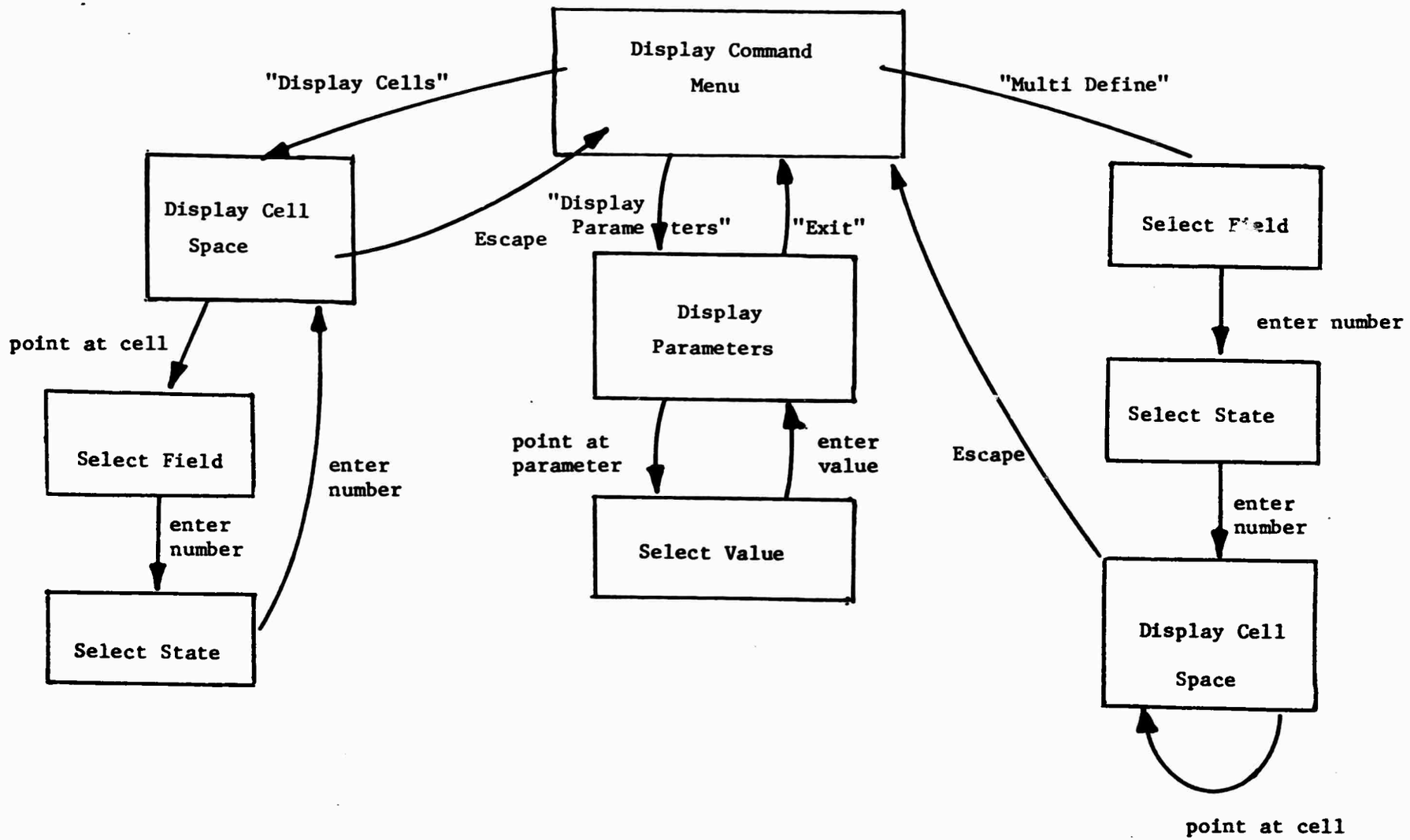
There are two commands available to resume the simulation or return to normal. "OVR" (over) starts the entire space transition over again. The box is removed from the offending cell, and if a micro-program was in progress, it is resumed. "CAN" (cancel) simply declares that no attempt will be made to resume, the box is removed and control stays at the keyboard. After either of these commands, all commands are once again legal. However, Back-up can not be performed after Cancel since the partially completed state transition computation destroys the previous time step state.

4.2 The Display Facilities

Control of sequencing of the various kinds of display is provided by the popular light button and state transition concepts. The initial display is a command menu allowing selection from a number of possibilities. All buttons result in the performance of some commands, many of which will entail display image transitions. Figure 4.5 illustrates the display transitions, which will become clearer as we proceed. Figure 4.6 shows the initial command menu.

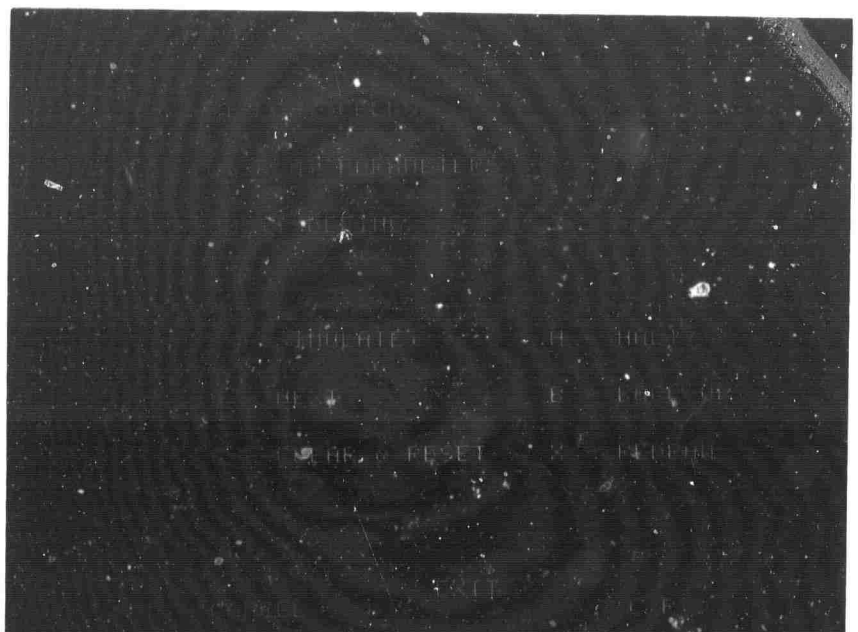
A limited amount of user editing is provided via two "control" light buttons near the lower edge of the screen: "CANCEL" and "C.R.". Light buttons are queued as they are selected and are processed when the "C.R." is selected. "C.R." may be considered an end-of-file, or carriage return, indication. Selecting the CANCEL button will clear the current command queue. Since in practice at most two or three commands are stacked at a time, more complex queue display or editing is not provided.

When a button is selected, the display provides positive indication of the "hit" by blanking all of the screen except the hit button for approximately



State Transition Diagram For Display Images

Figure 4.5



Command Menu
Figure 4.6

one-half second. The button is insensitive to the light pen during this period. This duration was empirically determined to be long enough to prevent unintended multiple hits of the same button, but not so long as to interfere with rapid sequential choice of light buttons.

Those commands which may only be given from the display are described here. They are 1) DISPLAY CELLS, 2) DISPLAY PARAMETERS and 3) MULTI DEFINE.

4.2.1 DISPLAY CELLS

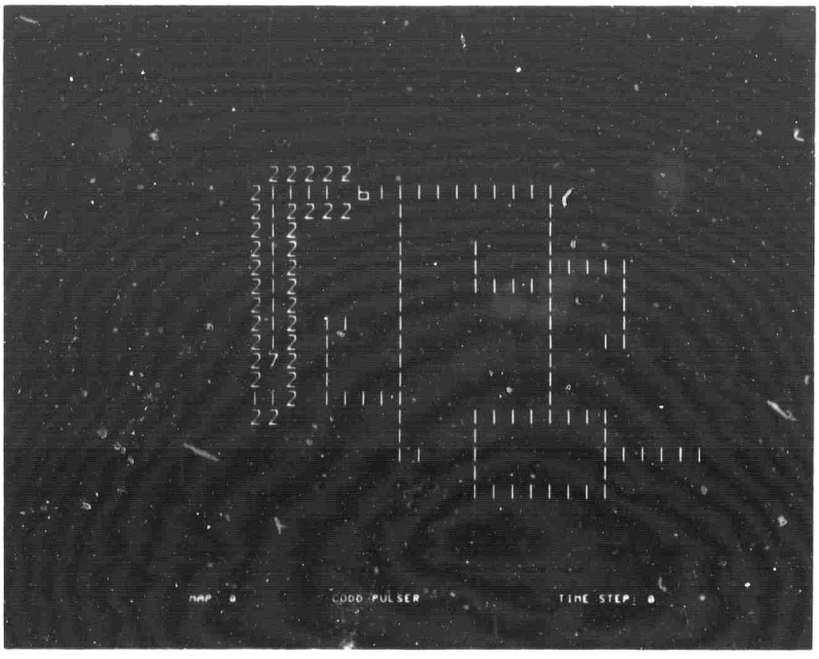
The basic display form for most monitoring activity is illustrated in Figure 4.7. The image consists of a) three identifiers shown along the lower edge, b) the actual cell space state under the current mapping, and c) a dot approximately at the middle of each of the four edges used to "move" the cell space state image.

The three identifiers are, in order:

1) The mapping number used to construct the current state display,
2) An arbitrary title defined by the user via a keyboard command, "TTL",
3) The time step of the simulation in progress.

These identifiers provide a simple and systematic means to document display results, for example, on film. But they also aid in reminding the user just where he is in the course of a simulation. In addition to their information content any of these identifiers serve as an "escape" light button for invoking the display command menu.

The cell state space is displayed in the major portion of the screen. Its outline and arrangement are determined by the corresponding commands described previously. The state of a cell is represented by a symbol or



Example Cell Space Display
Figure 4.7

graphic, one of 127 that the user can define by coding his desired symbols for the display character generator.

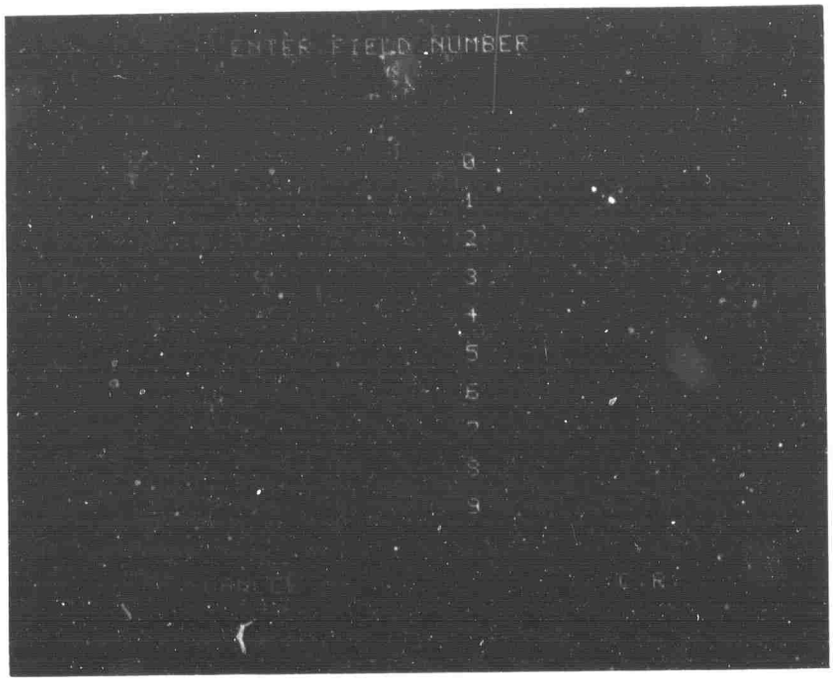
The availability of only 127 different symbols or graphics may seem like a strong restriction on the user's ability to monitor the state space. In practice, however, even in neural network simulations with quite large cell state spaces (i.e., describable by several integer and/or real variables) the judicious selection of a small set of mapping functions and heurististically meaningful graphics can actually enhance the experimenter's insight concerning the performance of the system.

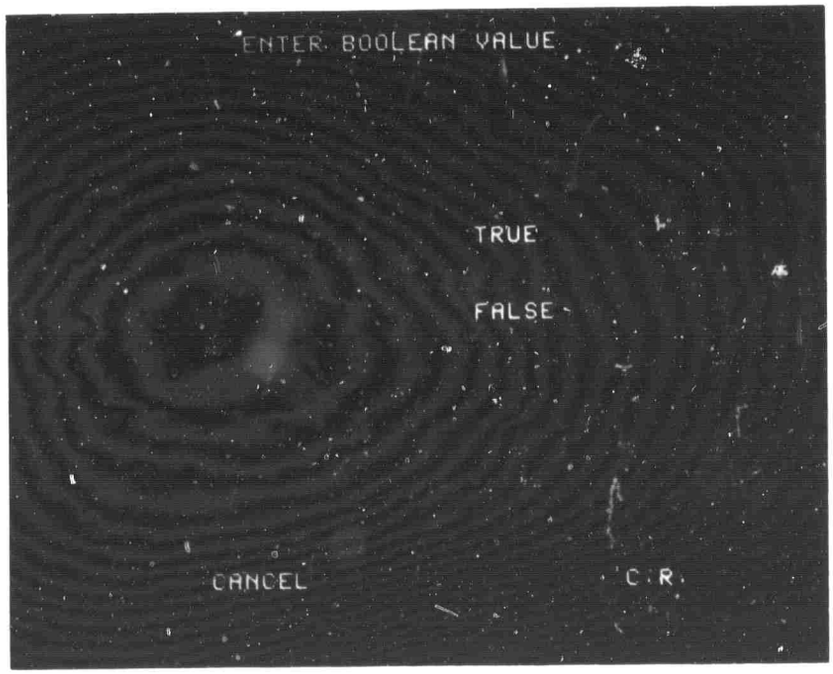
The DISPLAY CELLS data structure is arranged to enable the image to be viewed at any of four "scale factors" (x1, x2, x4 and x8) available on the display hardware. Particularly at the larger scale factors, only part of the image may fit on the screen. The four edge dots serve as light buttons causing the cell state image to move "underneath the display window" in the corresponding direction. The movement is discontinuous and incremental at a rate of approximately 2 cell sizes per second.

4.2.2 MULTI DEFINE

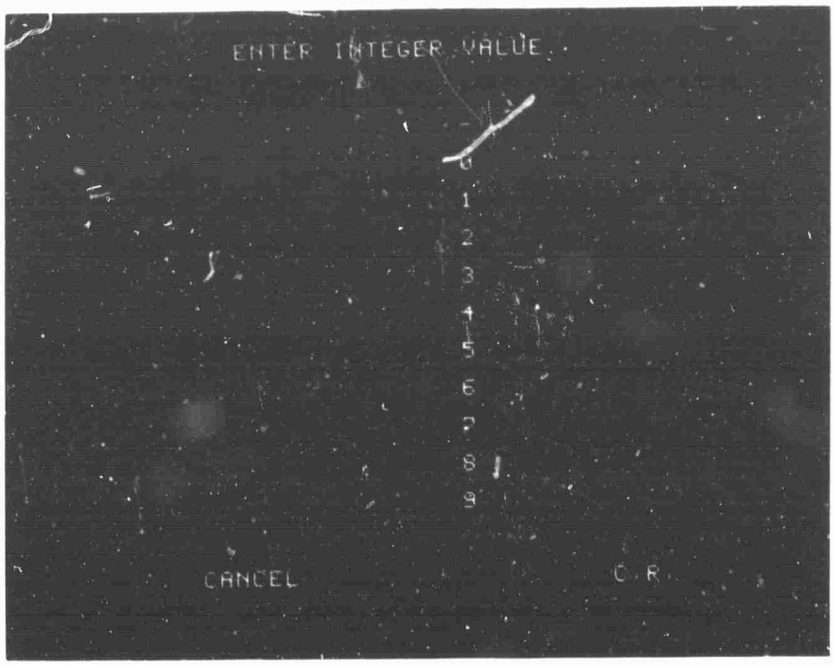
One way in which to make on-line changes to the cell space state is via the MULTI DEFINE command. This initiates a sequence of display transitions as follows: The display image of Figure 4.8 requests the user to enter, as an integer, the number of the field of the CELL data structure to be changed. This is done by selecting the appropriate sequence of light buttons terminated by the "C.R.".

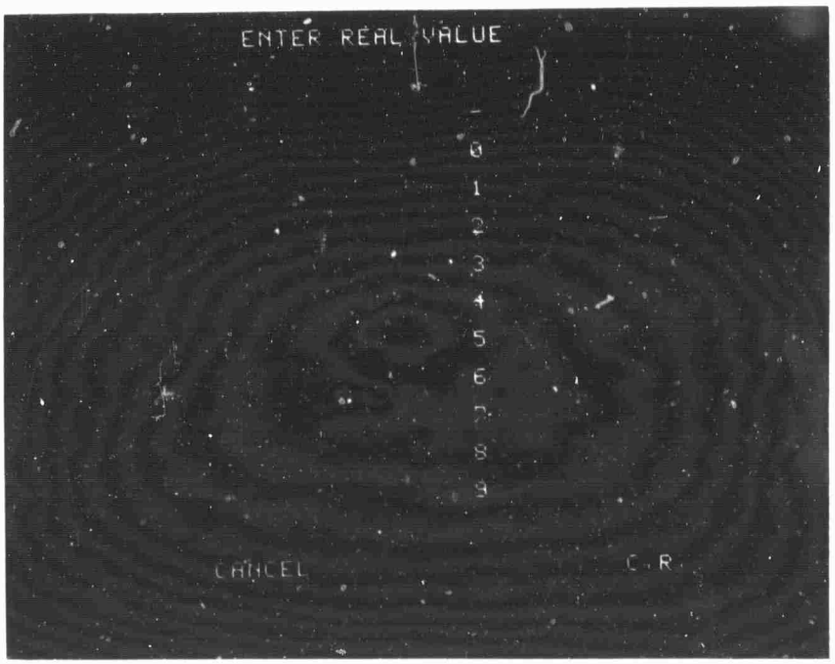
The next image requests a value for the field in the appropriate data type. Currently supported are the three data types BOOLEAN, INTEGER and REAL.





Data Entry Menus
Figure 4.8





Data Entry Menus
Figure 4.8 (Concluded)

The next image is the cell space display. Light pen hits on a cell will result in the indicated value replacing the value of the indicated field of the cell. Successive hits will result in the same assignment to multiple cells. The mode is terminated by escaping to the command menu.

A variation on this sequence occurs when displaying cells as a direct result of the DISPLAY CELLS command. Pointing at a cell will initiate the specify field-specify value sequence with the substitution performed for one cell only. Pointing to another cell initiates the sequence again.[1]

In both cases, at the time the cell state is updated a new image for the cell is computed via the current mapping function and substituted in the display data structure. Thus, if the current map is functionally dependent on the field changed, the user will get positive indication of the state change.

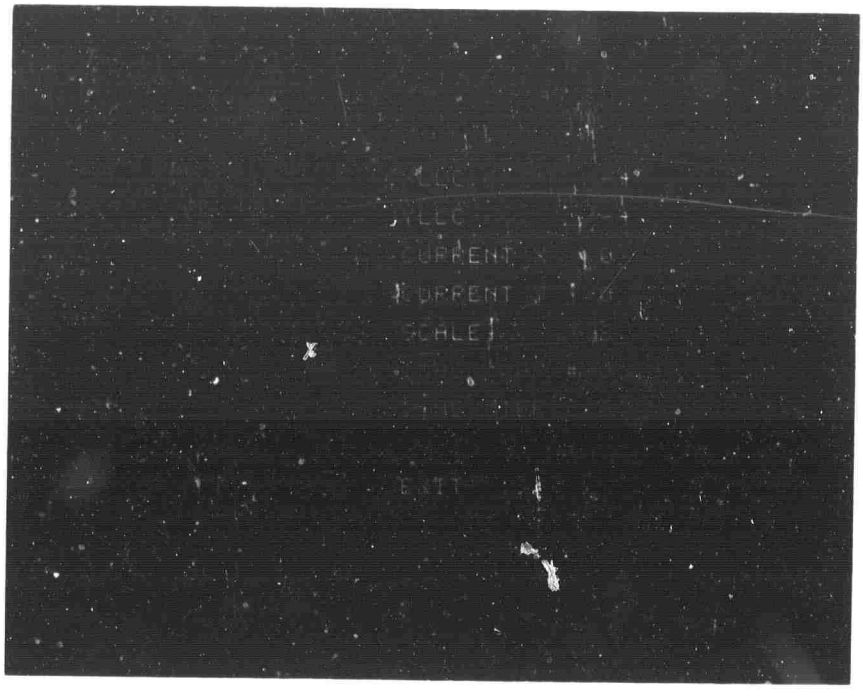
4.2.3 DISPLAY PARAMETERS

Hitting the DISPLAY PARAMETERS light button causes a transition to an image such as shown in Figure 4.9. The names of various system variables of interest to the user along with their current values are displayed. Those that can be changed by the user are light pen sensitive, while the others are not sensitive and are somewhat dimmer. A change is initiated by pointing to the name or its value. A new image requests a new integer value for the parameter.

"XLLC" and "YLLC" are the X and Y coordinates, respectively, of the

---

[1] Notice the nice manner in which the ability to manipulate the space state assignment on a cell by cell basis gives expression to the concept of embedding developed in Chapter 1.



Parameters Menu
Figure 4.9

lower left corner of the display window. Changing these values is an alternate way of moving the cell space portion of the DISPLAY CELLS image. "CURRENT X" and "CURRENT Y" are the coordinates of the most recent cell pointed to by the light pen or which currently is indicated by the box. "SCALE" is the hardware scale factor currently in use. Only values 1, 2, 4 and 8 are permitted for this variable. "IMAGE MAP #" and "TIME STEP" are for informational purposes only. "EXIT" causes a return to the main command menu.

5. APPLICATIONS, EVALUATION AND SUMMARY

This chapter illustrates the use of our simulation system in several application areas, evaluates the completeness and suitability of the system, and recommends areas for further development.

5.1 Applications

5.1.1 An Example from the Literature

To illustrate the use of the simulation system on a realistic problem, we have chosen an example from the literature which we will discuss in some detail. The work chosen was reported by Rochester, Holland, Haibt and Duba [14]. It is chosen because it requires many of the more unusual capabilities of our system, and is not too large to develop in a reasonable amount of space. The authors describe several simulation experiments performed to investigate and test the concept of Hebb concerning the formation of cell assemblies of neurons. We shall discuss one of their models called the FM or frequency model.

For maximum clarity, throughout the balance of this subsection we shall indicate the authors' specification in italics. The remaining text illustrates how to accomplish those requirements. The resulting program as a whole is shown in Figure 5.1.

There are 512 neurons arranged in a cylinder 16 units high and 32

```
DECLARE FM NAME;
DECLARE CMD USERENTRY;
DECLARE MAP0 MAPENTRY 0; DECLARE MAP1 MAPENTRY 1;
DECLARE MAP2 MAPENTRY 2; DECLARE MAP3 MAPENTRY 3;
DECLARE MAP4 MAPENTRY 4;
DECLARE INTEGER: XI,XIBAR,XJ,XJBAR,RI,RJ,XP,XPD;
DECLARE INTEGER : N,NN;
DECLARE FUNCTION: X,D;
DECLARE BOOLEAN:GEN;

DEFINESIZE <-16,-8> <-16,7> <15,7> <15,-8>;
DEFINEEDGE XWRAP;
DEFINENBHD <0,0>;
DEFINE AFFER BLOCK <COORD,REAL>;
DEFINE STATE ARRAY INTEGER SIZE 4;
DEFINE CELL BLOCK <AFFER,AFFER,AFFER,AFFER,
     AFFER,AFFER,SELECTINPUT,STATE>;
DEFINE EXTERNAL : < <<0,0>,0>,<<0,0>,0>,<<0,0>,0>,
     <<0,0>,0>,<<0,0>,0>,<<0,0>,0>,0,<0,3,7,49> >;
PARAMETER ( ISTATE 6) (FREQ 1) ( FREQBAR 2);
PARAMETER (FATIG 3) (R,4) (SELECT 7);

FM:       IF GEN;
*THIS COMPUTES THE RANDOM CONNECTIONS;
          LOOP N=1; 1; N $GT$ 0;
               NEWSTATE(N,1,1) = RANDI(17)-8;
               NEWSTATE(N,1,2) = RANDI(17)-8;
               NEWSTATE(N,2) = 0;
          ENDLOOP;
          GEN = FALSE;
        ELSE;

*THE NORMAL TRANSITION COMPUTATION;
NORMAL:   XI = CELLS(1,ISTATE,FREQ);
          XIBAR = CELLS(1,ISTATE,FREQBAR);
          RI = CELLS(1,ISTATE,R)*31/32 + (XI - XIBAR)
               $PS 2;
          XPD = 0; XP = 0;
          LOOP N=2; 1; N $GT$ 7;
               XJ = CELLS(N,ISTATE,FREQ);
               XJBAR = CELLS(N,ISTATE,FREQBAR);
               RJ = 31*CELLS(N,ISTATE,R)/32 +
                    (XJ - XJBAR) $PS 2;
               ST = NEWSTATE(N,2) * SQRTI(RI*RJ);
               ST1 = 31*ST/32 + (XI - XIBAR)*(XJ - XJBAR);
               NEWSTATE(N,2) = ST1/SQRTI(RI*RJ);
               XP =XP + RJ*XJ;
               XPD = XPD + ABS$(RJ*XJ);
          ENDLOOP;
```

FM CELL SPACE
FIGURE 5.1

```
                XP = 1.25*XP/XPD;
                DT1 = DI(NEWSTATE(ISTATE,FATIG),XP);
                NEWSTATE(ISTATE,FATIG) = DT1;
*THE CHECK FOR AN INPUT NEURON COMES HERE;
                IF NEWSTATE(SELECT) $EQ$ 0;
                     NEWSTATE(ISTATE,FREQ) = XI(XP,DT1);
                ELSE;
                     NEWSTATE(ISTATE,FREQ) = INPUT;
                ENDIF;
NORMAL1:        NEWSTATE(ISTATE,FREQBAR) = (31*XIBAR + XI)/32 ;
           ENDIF;
           RETURN;

CMD:       IF USERINPUT(3) $EQ$ "G";
                GEN = TRUE;
           ORIF USERINPUT(3) $EQ$ "S";
                NN = USERINPUT(6)@"INTEGER";
           ELSE;
                RETURN;
           ENDIF;
           USEROKAY = TRUE;
           RETURN;

*THE FIRST MAP CONVERTS WEIGHTS TO INTEGER J, 0 $LE$ J $LE$ 64;
MAP0:      GRAPHIC = (NEWSTATE(NN,2)+1.)*32;
           RETURN;
MAP1:      GRAPHIC = NEWSTATE(ISTATE,FREQ);
           RETURN;
MAP2:      GRAPHIC =NEWSTATE(ISTATE,FREQBAR);
           RETURN;
MAP3:      GRAPHIC = NEWSTATE(ISTATE,FATIG);
           RETURN;
MAP4: GRAPHIC = NEWSTATE(ISTATE,R);
           RETURN;

           ENDPROG;
```

F1 CELL SPACE
FIGURE 5.1 (CONCLUDED)

units around.

```
DEFINESIZE  [-16, -8] [-16, 7] [16, 7] [16, -7];
DEFINEEDGE XWRAP;
```

The neighborhood varies from cell to cell, being randomly generated at the start of an experiment and fixed thereafter. To accomplish this requires that the neighborhood be given as part of the data structure of each cell. Their model provided six inputs (i.e., six neighbors) for each cell. To this we shall add the cell itself, making a total of seven neighbors. Further, *this neighborhood has a distance bias that provides that cells within a distance of 8 units may be interconnected; those more distant may not.* To make a cell part of its own neighborhood represents a general neighborhood. It is accomplished by:

```
DEFINENBHD [0, 0];
```

Each input (afferent synapse) is characterized by a synapse weight that varies between -1 *and* +1. It is convenient to associate the neighbor coordinates and the synapse weight into a substructure as follows:

```
DEFINE AFFER BLOCK [COORD, REAL];
```

Each cell then contains six such substructures to give the six neighbors and their respective synapse weights.

The state of each cell is further characterized by four variables as follows:

FREQ	-	*FREQUENCY OF NEURON FIRING,* $0 \leq FREQ \leq 15$
FREQBAR	-	*average frequency,* $0 \leq FREQBAR \leq 15$
FATIG	-	*fatigue,* $0 \leq FATIG < 7$
R	-	*a smoothing variable,* $0 \leq R \leq 255$.

We shall define the STATE of each neuron thus:

```
DEFINE STATE ARRAY INTEGER SIZE 4;
```

Certain of the neurons are stimulated externally by an "outside" driving force. Thus we must declare a SELECTINPUT field.

Combining all of these, the type CELL is defined by:

```
DEFINE CELL BLOCK [AFFER, AFFER, AFFER, AFFER, AFFER, AFFER,
    SELECTINPUT, STATE];
```

For convenience we define the following names for referring to parts of the CELL data structure:

```
PARAMETER (ISTATE 8) (SELECT 7) (FREQ 1);
PARAMETER (FREQBAR 2) (FATIG 3) (R 4);
```

For external state we choose a neuron with itself for all six neighbors, all synapses of zero weight, and intermediate STATE:

```
DECLARE EXTERNAL:  [ [0,0], 0],  [ [0,0], 0],  [ [0,0], 0],
    [ [0,0], 0], [ [0,0], 0],  [ [0,0], 0],  [7, 7, 3, 0];
```

Since *the initial state requires the random generation of a neighborhood pattern for the space,* no INITIAL state will be declared. Rather the following code may be used to compute this information.

```
LOOP N; 1; N $GT$ 6;
   NEWSTATE(N,1,1) = RAND!(17) - 8;
   NEWSTATE(N,1,2) = RAND!(17) - 8;
   NEWSTATE(N,2)   = 0.;
ENDLOOP;
NEWSTATE(ISTATE)   = [7, 7, 3, 0];
```

The function RAND! (X) is a random number generator with integer parameter whose value is an integer I, $0 \leq I < X$. The expression in the example

limits connections to 16 by 16 square centered about a cell. It is only slightly more difficult to limit connections to a radius of 8 units about a cell. This code may be thought of as a transition function even though it is functionally independent of its arguments.

If the cell is to be an input cell, it will have a non-zero SELECTINPUT field. The following code assigns the cell firing frequency for the external input:

```
NEWSTATE(ISTATE, FREQ) = INPUT;
```

Finally, for a normal cell transition, the main transition function may be invoked. Since this section of code is lengthy, it is not repeated here. Refer to lines NORMAL to NORMAL1 of Figure 5.1. We shall not develop the coding for the cell behavior in detail since it is a straight-forward computation. Rather, let us call these three pieces of code *Init*, *Input*, and *Normal* respectively. They may then be combined as follows:

```
          DECLARE FM NAME;
FM:       IF GEN;
             Init
          ELSE;
             Normal
          IF NEWSTATE(SELECT) $NE$ 0;
             Input
          ENDIF;
          ENDIF;
          GEN = FALSE;
          RETURN;
```

To cause the BOOLEAN variable GEN to be TRUE we will define an input command

"U G"

via the following:

```
          DECLARE CMD USERENTRY;
CMD:      IF USERINPUT(3)$EQ$ "G";
             GEN = TRUE;
          ENDIF;
```

The general protocol is as follows:

1) Compile and load system,
2) Type "U G" command to set variable GEN,
3) Type N command to cause transition to establish the initial state.
4) This state may then be kept, if desired, to be used later.

This completes the coding of the simulation proper. There remains to provide the mapping functions for displaying the state. Four obvious maps display the four INTEGER variables comprising the STATE:

```
MAP1:          GRAPHIC = NEWSTATE(ISTATE, FREQ);
               RETURN;
MAP2:          GRAPHIC = NEWSTATE(ISTATE, FREQBAR);
               RETURN;
MAP3:          GRAPHIC = NEWSTATE(ISTATE, FATIG):
               RETURN;
MAP4:          GRAPHIC = NEWSTATE(ISTATE, R);
               RETURN;
```

To display the synapse weights requires six more maps; these can be

collapsed into one map by using a "U" command to select the synapse field:

```
MAPØ:          GRAPHIC = 32*(1. + NEWSTATE (NN,2));
               RETURN;
```

Augment the user entry as follows:

```
               ORIF USERINPUT(3) $EQ$ "S";
               NN = USERINPUT(6);
               USEROKAY = TRUE;
```

This completes the program. The entire program is shown in Figure 5.1.

We emphasize that the resulting program should not be construed as the ideal or unique implementation of the neural model. It is one solution that can serve as a framework for following-up on the reported work. Considerable development in many directions is possible depending on the interests of the investigator. It demonstrates, however, the relative ease with which cellular models may be accomplished in our system.

5.1.2 Current Work

James Mortimer (Logic of Computers Group) is currently investigating a class of neurophysiological models that have been implemented on the simulation system.[1] He has defined a cellular model characterized roughly as follows:

1) The model consists of 400 cells in a rectangular sheet, each cell corresponding to a small anatomical region of the cortex.

2) Each cell represents a population of each of five distinct neuron types lying within each anatomical region.

---

[1] Because the Run Time Environment preceded the language compiler, Mortimer's programs are implemented in a mixture of FORTRAN and assembly language. His implementation is completely compatible with the RTE described here.

3) The state of a cell is the frequency of firing of each neuron type.

4) A uniform neighborhood rule is used with the number of neighbors counted in the dozens.

5) Input (external electrical stimulation) is provided to each of the neuron types.

6) A quasi-linear, frequency-modulated model of neuronal behavior is used to define transition behavior.

Mortimer's goals include, 1) finding a physiologically acceptable set of model parameters that allows spontaneous, stable activity to persist in all neuron types, 2) demonstrating that alternating excitatory and inhibitory inputs induce an alternating pattern of high and low frequency firing, and 3) investigating the hypothesis that there exist distinct excitability states in the cortex in which incoming signals are processed in fundamentally different ways.

Roger Weinberg and Erik Goodman, (Logic of Computers Group) are modifying a program of Weinberg and Berkus [17] which simulates the known major chemical processes of a single cell in order to run multiple cell simulations under the present system. They hope to explore several adaptive and growth processes, such as competition for environmental resourses, conditions for normal versus explosive (cancerous) growth behavior, and conditions for competitive versus cooperative interaction.

5.1.3 Related Problem Areas

We should like to point out two additional problem areas that have not been important motivators in the development of our simulation system but in which, none-the-less, our system has potential applications. We

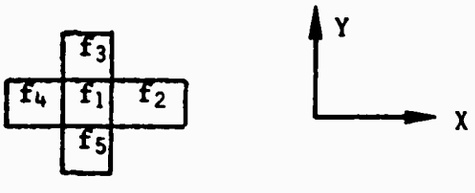
introduce them here to further demonstrate the broad applicability of the concepts of cellular models and hence of programming tools embodying those concepts.

Relaxation methods used for solving partial differential equations fit well within the framework of this simulation system. Consider the case of the two-dimensional Laplace partial differential equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

Each cell of the model may represent a small area (of size delta-X by delta-Y) and the state of each cell represents the value of the function at the center of the cell. Delta-X and delta-Y thus gives the fineness of the approximating mesh. The equations are rewritten in terms of finite differences and the neighborhood is defined to be large enough to include the cells needed to compute these differences. In this case, the familiar five cell neighborhood suffices.

Suppose we have the local states as shown below:



Then second order partials are approximated by the second order differences

$$\frac{\partial^2 f}{\partial y^2} \cong \frac{\nabla^2 f}{\nabla y^2} = (f_3 - f_1) - (f_1 - f_5)$$
$$= f_3 + f_5 - 2f_1$$

and

$$\frac{\partial f^2}{\partial x^2} \cong \frac{\nabla^2 f}{\nabla x^2} = (f_2 - f_1) - (f_1 - f_4)$$
$$= f_2 + f_4 - 2f_1$$

The Laplace equation requires that these sum to zero:

$$(f_2 + f_4 - 2f_1) + (f_3 + f_5 - 2f_1) = 0$$

or

$$f_1 = (f_2 + f_3 + f_4 + f_5)/4$$

The transition function then consists of computing the new state of a cell as the average of its neighbors. Successive time steps are computed until the space state converges sufficiently to stable values.

The technique generalizes to higher order and non-linear partial differential equations and in general our simulation system should be a suitable vehicle for solving such equations whenever relaxation methods are appropriate. We note that forcing functions (input) and boundary conditions (external state) are easily accomplished in the system.

Ulam [18] and others[1] have worked with certain problems concerning the growth of patterns. Starting from a regular arrangement of cells (such as squares of equilateral triangles) and some initial configuration of non-blank cells, additions to the current configuration are defined for successive time steps. Growth rules typically consist of local rules or conditions for a non-blank state cell to "extend" to an adjacent cell. For example, Ulam discusses the following rule for a two-dimensional plane of squares:

> "Our growth is in the plane subdivided into regular squares. The starting configuration may be an arbitrary set of (closed) squares. The growth proceeds by generations in discrete intervals of time. Only the squares of the last generation are "alive" and able to give rise to new squares. Given the n^{th} generation, we

---

[1] See also work by J.C. Holladay and R.G. Schrandt.

define the $(n+1)^{th}$ as follows: A square of the next generation is formed if

a) it is contiguous to one and only one square of the current generation, and

b) it touches no other previously occupied square except if the square should be its "grandparent". In addition:

c) of this set of prospective squares, of the (n+1) generation satisfying the previous condition, we eliminate all those that would touch each other. Agian there is an exception for those squares that have the same parent; these are allowed to touch."

The variations on such growth rules are bountiful, the richness and apparent complexity of the resulting figures is substantial, and the difficulty in obtaining analytical results is frustrating. After studying the growth of figures from temporal and spacial points of view, Ulam asserts that

> "the geometry of objects defined by recursions and iterative procedures deserves a general study - they produce a variety of sets different from those defined by explicit algebraic or analytical expressions or by the usual differential equations."

We point out the intimate relation of these figures to the cellular model and simulation capabilities developed here, and suggest our system is a suitable tool for studying these problems.

5.2 Evaluation

The primary consideration in the evaluation of this system is the extent to which it presents a language and programming environment that simplifies and shortens the effort required to initiate a simulation. We know of no conclusive way in which to demonstrate that our system is optimal in this respect. However, we are confident that we can substantiate that it is better in this respect than previously available languages

and facilities.

It is clear that to do a simulation of a cellular space in some other general purpose language must require at least as much effort as in the Cellular Space Language. The computational aspects of state transition routines can in general be met only by the full resources of an algebraic language. (Clearly if we restrict our attention to a special class of cellular simulations, e.g., Von Neumann-like spaces, more special purpose languages such as simple tabular languages might be a more natural and compact representation.) But in addition to providing the transition function, the "cell space properties" must still be provided in other languages. The needed concepts and data structures for the space and neighbor specification, for example, simply are not available in existing languages and must be synthesized afresh for each application. To illustrate, in the simulation performed by Flanigan approximately forty lines of Fortran coding are required to define his neighborhood relationship, while in the author's system this is accomplished by a single declaration. Moreover, any change in the neighborhood relation is correspondingly easier to effect.

5.3 Extensions

There are several areas in which the current systems design could be expanded, either to enlarge the range of cellular models that can be conveniently accommodated or to increase its general suitability as an interactive programming system.

The most important constraint of the present system that ought to be relaxed is the fixed number of dimensions of cellular spaces. Few, if any, changes are required to the specification of the Cellular Space

Language itself in order to handle higher dimensional spaces. Most changes would come in the implementation of the Run Time Environment, but even these are quite simple to introduce. The reason that this generalization has not been included from the beginning lies more in the area of the heuristic utility of the resulting system than in the difficulty of implementation. It is not at all clear what kind of display facilities beyond that already provided would be sufficient to permit the experimenter to adequately cope with the higher dimensionality. The display problem is distinct from most of the three dimensional graphic research being performed today since removal of hidden parts or perspective views are generally not relevant to the simulation problem. We conjecture that the ability to display the state of arbitrary planes or "lamina" through such a higher dimensional cell space is the best that can be provided. Such displays do not extend the heuristic support of the system to any considerable extent.

For certain classes of simulations, the state space of the cell (or of particular substates of a cell) is sufficiently small that the mapping functions may be used to define a one-to-one relationship between that (sub)state and the set of graphics used to display the (sub)state. For such cases it would be particularly useful if the experimenter could use a menu consisting of the set of possible graphics to <u>assign</u> state values as well as display them. Indeed, even in problems with a many-to-one mapping, the graphics may be sufficiently characteristic of the state that it would be adequate, for the purposes of the simulation, to use the graphics to assign a particular state (from among those which map into that graphic). A possible implementation would provide for the

specification of "inverse maps", which would be used quite similarly to maps, except that their domain would be the set of graphics and their range the cell state space.

Thomas Schunior (Logic of Computers Group) has suggested an idea for making more efficient use of core storage in many simulations. He would partition the state of each cell into a private part and a public part. The *private* part of a cell state space is that sub-state space which does not functionally take part in the state transition of any other cells. The *public* part consists of the remainder of the state space. The private state information need not be stored in the redundant form adopted earlier since it clearly will not be accessed after being changed. For the same reason, this data could even be kept on a bulk storage device and streamed in and out of core in synchronism with the algorithm for scanning the cell space during a transition. The obvious advantage is that the size of simulation that can be represented with a given amount of core storage is increased. Moreover, experience with the computational requirements of typical simulations indicates that good advantage could be taken of bulk storage in this manner without introducing input/output delays due to accessing the bulk storage.

A more subtle issue is raised by the following consideration. In certain biologically oriented models, it is quite natural to think of a cell as *causing* a change of state in some neighboring cell. Formally, this is easily accomplished in our cellular model; though it may entail enlarging the neighborhood of each cell so that the transition function might "look to see" if some neighbor would be causing a change. But the

point of view is somewhat awkward. We suggest a notion of output neighborhood which gives the range of influence of a cell. Our previous notion of neighborhood might then be better called the input neighborhood or the domain of dependency of the cell. The obvious formal problem is that any given cell may be in the output neighborhood of several cells which are attempting to cause mutually exclusive changes. (Holland faced a similar problem in defining the path building rules for his ICC's.) But where resolution procedures can be introduced, or where it can be shown that inconsistent behavior can not be generated, such an expanded model may be a better framework for study, in the sense that it more closely mirrors the way certain systems are conceptualized.

We recognize that the subscript notation for referring to the neighbors of a cell is hardly the most intuitive manner of doing so. Whenever possible, one would like to employ mnemonics, such as"RTCELL" for the cell to the right. It is clear, however, that because of the freedom in defining the neighborhood relationship and because the neighborhood can even change dynamically, no naming convention can possibly be both suitable to implement and heuristically useful to most people. The best solution to this dilemma is probably the inclusion of a generalized parameter facility in the Cell Space Language. Such a facility would permit a user to define, for example, "TOPCELL" to mean "CELLS(3)" in order to refer to a particular neighbor more conveniently, and then to be able to write an expression such as "TOPCELL(1,NN)" to refer to substates of the cell. Such a facility is useful in almost any language, not just a language for cellular simulations. Accordingly, we have not included a parameter

facility within our language.

A potentially valuable extension not included for similar reasons is the provision of a run-time symbol table. This, together with an appropriate run-time interpreter or "debugger", would be a valuable tool in the testing of new models.

We speculate that a valuable adjunct to any interactive system that purports to enhance the heuristic capabilities of its users is a facility for maintaining an automatic history of all actions taken by the users over some prescribed period of time. This would permit them to better keep track of activities during a complex exploration, and even permit the reconstruction at a later time of the exact circumstances of a space behavior whose significance may not have been realized when it occurred. Whether the utility is worth the cost of implementation we leave for others to explore.

5.4 Summary

Regular networks of similar interacting components constitute an important class of models in many disciplines, from automata theory to parallel computer systems to biological systems. A number of significant applications of such models have been surveyed and drawing upon that and related material, a programming system has been proposed for conveniently simulating such models. Careful attention has been given to setting forth the motivations behind the final proposed form. Lastly, several applications developed on the system have been discussed in order to integrate the development and demonstrate the utility of the system.

The final measure of the success of this effort must rest with

those who will now attempt to approach the system as a tool for doing their simulations. Two such efforts are well underway. To these and others go the task of further validating and extending the premise of this thesis: that an integrated and viable programming system can be created for simulating a broad range of cellular spaces.

REFERENCES

1. Burks, Arthur W., Von Neumann's Self-Reproducing Automata, Technical Report 08226-11-T, Computer and Communication Sciences Dept., University of Michigan, June 1969.

2. Codd, Edgar F., Propagation, Computation and Construction in Two-Dimensional Cellular Spaces, Ph.D. Dissertation, University of Michigan, 1965, also as Cellular Automata, Academic Press, 1968.

3. Comfort, W. T., "A Modified Holland Machine," in Proceedings-Fall Joint Computer Conference, 1963.

4. Crichton, J. W., and Holland, J. H., "A New Method of Simulating the Central Nervous System Using an Automatic Computer," Technical Memorandum 2144-1195-M, The University of Michigan, March 1959.

5. Evans, George W., II, Graham F. Wallace, Georgia L. Sutherland, Simulation Using Digital Computers, Prentice Hall, 1967.

6. Flanigan, Larry Karl, A Cellular Model of Electrical Conduction in the Mammalian Atrioventricular Node, Ph.D. Dissertation, University of Michigan, 1965.

7. Finley, Marion, Jr., An Experimental Study of the Formation of Hebbian Cell-Assemblies by Means of a Neural Network Simulation, Ph.D. Dissertation, University of Michigan, 1967.

8. Hebb, D. O., Organization of Behavior, John Wiley & Sons, New York, 1949.

9. Holland, J. H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously," pp. 108-113 of the Proceedings of the 1959 Eastern Joint Computer Conference, Institute of Radio Engineers, 1959. Also in Essays on Cellular Automata.

10. ___, "Iterative Circuit Computers: Characterization and Resume of Advantages and Disadvantages," pp. 171-178 of Microelectronics and Large Systems. Edited by Mathis, Wiley, and Spandorfer. New York: Spartan Books, 1965. Also in Essays on Cellular Automata.

11. ___, "Universal Spaces: A Basis for Studies of Adaptation," pp. 218-230 of Automata Theory. Edited by E. R. Caianiello. New York: Academic Press, 1965.

12. Michigan Algorithm Decoder (MAD) Manual, University of Michigan Press, Ann Arbor, Michigan, August 1966.

13. Parnas, David L., "On Simulating Networks of Parallel Processes in Which Simultaneous Events May Occur," Comm. A.C.M. 12, 9 (September 1969) pp. 519-531.

14. Rochester, N.; J.H. Holland; L.H. Haibt; W.L. Duda., "Tests on a Cell Assemble Theory of the Action of the Brain, Using a Large Digital Computer," in Transactions on Information Theory, IRE, IT-2, No. 2.

15. Von Neumann, John, Theory of Self-Reproducing Automata. Edited & Completed by Arthur W. Burks, University of Illinois, Urbana, Illinois, 1966.

16. Wagner, Eric G., "On Connecting Modules Together Uniformly to Form a Modular Computer." in 1965 IEEE Conference Record on Switching Circuit Theory and Logical Design, October 1965.

17. Weinberg, Roger and Berkus, Michael, Computer Simulation of a Living Cell, Technical Report 01252-2-T, Computer and Communication Sciences Department, University of Michigan, May 1969.

18. Ulam, Stanislaw, "On Some Mathematical Problems Connected with Patterns of Growth of Figures" in Mathematical Problems in the Biological Sciences, pp. 215-224. Proceedings of Symposia in Applied Mathematics, Vol. 14, Providence, Rhode Island: American Mathematical Society, 1962.

19. Yamada, H., and S. Amoroso, "Tessellation Automata," (Unpublished) April 1968.

APPENDICES

TABLE OF CONTENTS

LIST OF FIGURES

APPENDIX A. SUMMARY OF THE LANGUAGE

This section is a collection of figures summarizing the major features of the simulation language. In addition to presenting a descriptive syntax for the language, several figures detail the operators and associated data types that are defined in the current implementation. In these latter figures the following abbreviations are used:

B	BOOLEAN variable or constant
I	INTEGER variable or constant
Ic	INTEGER constant only
R	REAL variable or constant
T1	TEXT variable or constant of exactly one character

Strictly speaking the atoms = and ! are not operators in the formalism used here. However, it is useful to think of = as an operator that accepts any two arguments of the same type. In addition, the appropriate conversion will be performed when assigning as INTEGER value to a REAL variable, or a REAL value to an INTEGER variable.

<declare part> → DEFINE λ ARRAY <type> SIZE <integer>;

→ DEFINE λ BLOCK [<type-list>];

→ DEFINE λ UNARY <type>, <type>;

→ DEFINE λ BINARY <integer>, <type>, <type>,<type>;

→ ENDOPR;

→ DEFINENBHD {<coordinate>};

→ DEFINESIZE $\{\text{<coordinate>}\}_{3}^{\max}$;

→ DEFINEEDGE $\begin{bmatrix} \text{XWRAP} \\ \text{YWRAP} \\ \text{TORUS} \\ \lambda \end{bmatrix}$;

→ DECLARE λ NAME;

→ DECLARE λ MAPENTRY <integer>;

→ DECLARE λ USERENTRY;

→ DECLARE λ PRETRANSENTRY;

→ DECLARE λ POSTTRANSENTRY;

→ DECLARE <type>: <λ-list>;

→ PARAMETER { (λ $\{,\}_{0}^{1}$ <integer constant>) };

Declarations of the Language

Figure A.1

```
<executable> → IF <exp>;
             → ORIF <exp>;
             → ELSE;
             → ENDIF;
             → LOOP     <assignment>; <exp>; <exp>;
             → ENDLOOP;
             → <assignment>;
             → CONTINUE;
             → EXECUTE <exp>;
             → GOTO <exp>;
             → RETURN;
             → ENDPROG;
```

Executable Statements of the Language
Figure A.2

<assignment> → <left des> = <exp>

<left des> → λ

→ λ (<exp list>)

<exp list> → <exp>

→ <exp list>, <exp>

<exp> → <exp> θ <exp>

→ φ <des>

→ <des>

<des> → <left des>

→ (<exp>)

→ λ! (<exp list>)

→ λ!

→ <lsv>

<des> → (<assignment>)

<lsv> → [<exp list>]

Syntax of Assignment Statement
Figure A.3

GRAMMAR

<STMT>	=	<BEG*>	<ASSIGNMENT>	
<STMT>	=	<BEG*>	<EXP>	
<ASSIGNMENT>	=	<LEFT DES-=*>	<EXP>	
<LEFT DES-=*>	=	<LEFT DES>	=	
<LEFT DES>	=	<LAMBDA*>		
<LEFT DES>	=	<LAMBDA-(*>	<EXP LIST>	)
<LAMBDA-(*>	=	<LAMBDA*>	(	
<LAMBDA*>	=	LAMBDA		
<EXP LIST>	=	<EXP>		
<EXP LIST>	=	<EXP LIST-,*>	<EXP>	
<EXP LIST-,*>	=	<EXP LIST>		
<EXP>	=	<EXP-THETA*>	<EXP>	
<EXP>	=	<DES>		
<EXP-THETA*>	=	<EXP>	THETA	
<DES>	=	<PHI*>	<DES>	
<PHI*>	=	PHI		
<DES>	=	<LEFT DES>		
<DES>	=	<(*>	<EXP>	)
<DES>	=	<LAMBDA-!-(*>	<EXP LIST>	)
<DES>	=	<(*>	<ASSIGNMENT>	)
<DES>	=	<LSV>		
<LAMBDA-!-(*>	=	<LAMBDA-!*>	(	
<LAMBDA-!*>	=	<LAMBDA>	!	
<(*>	=	(		
<LSV>	=	<LSB*>	<EXP LIST>	RSB
<LSB*>	=	LSB		
<DES>	=	<LAMBDA-!*>		

Note:	THETA	-	binary operator
	PHI	-	unary operator
	LSB	-	left structure bracket, [
	RSB	-	right structure bracket,]

Augmented Operator Grammar for Statements and Expressions
Figure A.4

Procedural*	Cell Space Related
INTEGER	CELLS
REAL	COORD
BOOLEAN	CELL
LABEL	INITIAL
FUNCTION	EXTERNAL
TEXT	INPUT
DECLARE	SELECTINPUT
DEFINE	DEFINENBHD
ARRAY	DEFINESIZE
SIZE	DEFINEEDGE
BLOCK	XWRAP
UNARY	YWRAP
BINARY	TORUS
ENDOPR	LOCATE
ENDPROG	NAME
IF	USERINPUT
ORIF	USEROKAY
ELSE	
ENDIF	
LOOP	
ENDLOOP	
GOTO	
CONTINUE	
EXECUTE	
RETURN	

(* The operators of Figures A.6 and A.7 are also reserved atoms.)

Reserved Atoms - Keywords

Figure A.5

Operators	Type of Arguments		Type of Result
1. + - * / P	I	I	I
	I	R	R
	R	I	R
	R	R	R
2. EQ NE GT GE LT LE	I	I	B
	I	R	B
	R	I	B
	R	R	B
3. AND OR	B	B	B
4. $BITAND$ $BITOR$ $BITXOR$	I	I	I
	T1	I	T1
	I	T1	T1
5. RS LS (right and left shift)	I	Ic	I
	T1	Ic	T1
6. EQ	I	T1	B
	T1	I	B
	T1	T1	B

Binary Operators
Figure A.6

Operators	Type of Argument	Type of Result
1. - ABS$	I R	I R
2. NOT$	B	B
3. BITNOT$	I	I
4. FIX$	R	I
5. FLOAT$	I	R

Unary Operators
Figure A.7

RS LS $BITAND$ $BITOR$ $BITXOR$	70
P	60
* /	50
+ -	40
EQ NE GT GE LT LE	30
AND	20
OR	10

Precedence Values For the Binary Operators
Figure A.8

APPENDIX B. COMPILER IMPLEMENTATION

The compiler is relatively conventional in organization. It consists of a lexical processor for extracting atoms from the input stream, followed by a syntax driven analyzer which produces an intermediate text form commonly called triples. A second pass processes the triples and as its output produces source code in 1800 assembler format. This source must then be assembled via the 1800 monitor system.

B.1 Pass 1 - Source Code Reduction

B.1.1 Lexical Scan

Each atom is assigned a lexical type based on the lexicographic characteristics of the atom. The set of lexical types constitutes the terminal vocabulary of the syntactic parser. The complete set of lexical types and their defining characteristics is presented in Figure B.1.

The lexical parser is basically a finite state acceptor that reads individual characters from the input until a terminating state is reached. Depending on the final state, the terminating character may be included in the atom being assembled, or saved to start the succeeding atom. Each character is assigned a class code (running from 0 to 6); this code actually drives the finite state acceptor. To make this organization feasible we have minimized the multiple use of characters, e.g., period ".", in different contexts while attempting to avoid an overly artificial lexical appearance. Characters are classified as follows:

Binary Operators θ	+ - * / and λ-atoms defined as binary operators, e.g., EQ
Unary Operators	- and λ-atoms defined as unary operators, e.g., ABS$
Punctuation - with each character a distinct catagory in itself	[ ] ( ) , ! = : ;
Keywords	Each predefined, e.g., IF, LOOP, DECLARE, REAL
All others	λ-atoms

Lexical Types of the Language
Figure B.1

Illegal Characters	0
Numerics	1
Alphabetics	2
Space and Newline	3
Quote	4
Punctuation	5
Other Legal Characters	6

After each atom is collected, its lexical type is determined from its symbol table entry if the atom is predefined or has occurred before; otherwise the lexical type is λ. The output of the lexical scan is a sequence of pointers to the symbol table entries for the atoms of an input line. Entries are created as needed for new atoms.

B.1.2 Syntactic Parsing

Of the many possibilities for syntactic recognition of expressions, we choose to modify a technique developed by Gries [4] employing Augmented Operator Grammars. Gries' technique has the following advantages over most other syntax recognizers:

1) It has the speed advantages of left-to-right without back-up, bottom-top techniques used with the operator and precedence grammars.

2) The recognition algorithm gives the symbol to which to reduce a "handle", as well as detecting the handle proper.

3) The organization of the recognizer permits extra-syntactic activities, such as triple generation, to occur at many points in handle reduction. (This is important in the handling of LSVs in our compiler. For example, a triple is generated by the atom "<" which marks the start of an LSV.)

Our modification of the AOG technique concerns the handling of binary operators. In the operator grammars, the precedence of binary operators is indicated in the grammar itself. Consider the "typical" grammar

```
<term>    →  λ
          →  <term> * λ
<exp>     →  <term>
          →  <exp> + <term>
```

where λ is a terminal class consisting of variable names in the language. The higher precedence of multiplication (*) over addition (+) is explicitly part of the syntax. This results in a syntactic description that becomes longer as each new operator is introduced. Moreover, a new parser must be logically generated each time a new operator is introduced, since it involves a change in the syntax. Floyd [3], for example, would have to calculate new f and g functions. In practice this is often not too difficult to do because the high degree of regularity among the grammar rules is reflected in the recognizer. The fact remains, however, that the augmentation must be explicitly accommodated by the parser.

In the modified form employed by this author, the following (simplified) syntactic description is used:

```
<exp>     →  λ
          →  <exp> θ <exp>
```

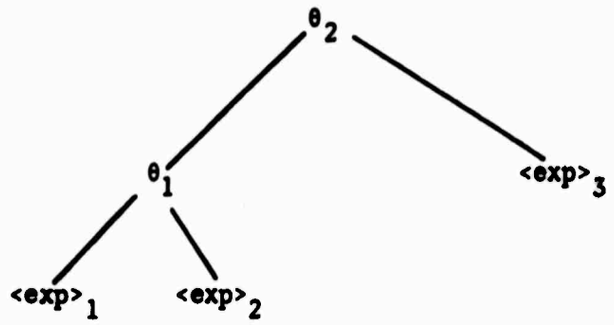
where θ represents the class of binary operators, e.g., + * /. Thus θ is a terminal class of the grammar.

It is clear that such a grammar is, of itself, ambiguous; the precedence information is lacking. This information is essential to be able to parse the following:

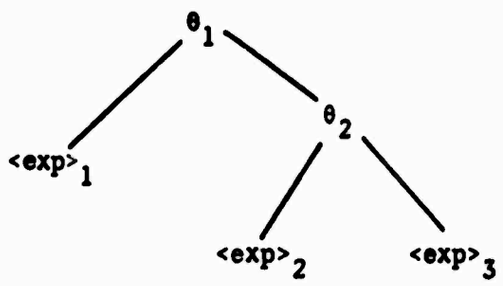
$$\langle exp\rangle_1\ \theta_1\ \langle exp\rangle_3\ \theta_2\ \langle exp\rangle_3$$

Let f be a function from the class of binary operators θ to the natural numbers, f: θ → N. Parse the sentential form by associating first on the

left if $f(\theta_1) \geq f(\theta_2)$. This yields:



If $f(\theta_1) < f(\theta_2)$, then associate first to the right. This yields:



The function f: $\theta \rightarrow N$ corresponds exactly to the usual notion of arithmetic precedence for the binary operators. Moreover, defining a new binary operator consists of declaring some atom α to be an element of the class θ and associating with it the value $f(\alpha)$.

Another departure from conventional parsing techniques occurs in the handling of literal structured variables. In order to provide additional information to the routines that will expand triples, we take advantage of the fact that extra-syntactic actions can occur each time an initial segment of a production is reduced. This enables us to provide more efficient run

time code than can be done with the result of an operator analysis.

In order to facilitate the manipulation of cell states by the space state data management routines, we must maintain the data structure of each cell as a contiguous block of memory. In addition, data structures which allow "pointers" impose an overhead in memory requirements that is unjustifiable since all cells of the space must have the same data structure.

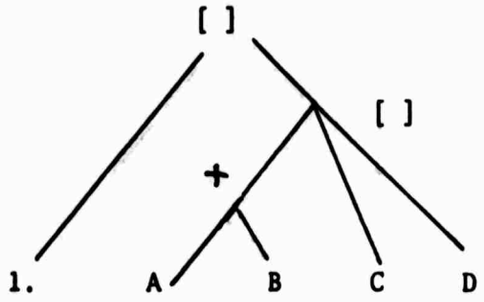
The requirement of a contiguous, non-pointer data structure means that if LSVs are parsed and interpreted in the usual manner, then an excessive and unnecessary amount of physical data movement will result. Consider the following:

```
DEFINE INT3 BLOCK [INTEGER, INTEGER, INTEGER];
DECLARE INTEGER: A, B, C, D;
DEFINE QRST BLOCK [REAL, INT3];
```

The expression

```
[1., [A+B, C, D]]
```

might be parsed as follows:



The usual inside-out method of generating code results in the following sequence of operations:

1) Select a temporary location, compute A+B and save it there.

2) Select a temporary storage area for the inner block and move A+B, C and D into place.

3) Select a temporary storage area for the outer block, and move 1. and [A+B,C,D] into place.

Clearly much of this data movement is unnecessary if, at the bottom of the parsing tree, one can know where the data will eventually be needed at the higher level(s). With such information, the following shorter sequence of operations could result:

1) Select a temporary for the outer block.

2) Compute A+B and save into place.

3) Move 1., C, and D into place.

The necessary information can be extracted and saved in the triple storage in the course of parsing LSV structures. The details of this process will be the subject of further work.

B.1.3 Statement Processor

Generally a statement is made up of a keyword followed by an expression. Recognition of the keyword causes a dispatch to the appropriate routine for handling that language construct. These routines may themselves directly output triples and may also call on the expression parser to "parse out" an expression and output triples. Detection of global errors, such as unbalanced LOOP... ENDLOOP and IF...ENDIF pairings, are performed by these routines.

B.1.4 Intermediate Program Representation

The result of pass one processing is represented by three kinds of tables: the triple table, the symbol table, and the structure table.

The triple table represents the executable portion of the program. The symbol table contains entries for all atoms used in the program, together with their properties or attributes. The structure table represents the data structures defined by the program, either explicitly or implicitly.

B.2 Pass 2 - Object Code Generation

The second pass over the source program - now in its intermediate form - produces as its output the machine language program to be executed. This "object" program is represented as 1800 Assembler source coding to be subsequently assembled.

The symbol table is first examined for the information needed to build the transition routine control block. This control block is the primary linkage between the compiler and the Run Time Environment. Its structure and interpretation is discussed C.2.3.

Next is the processing of the triple table. Each triple is viewed as a macro call which directs the action of a simple macro expander. Macro definitions are of two types: symbolically defined and built-in.

The symbolically defined macros are written in a simple macro language. The definition set is read in and stored for interpretation at the beginning of pass 2. Local object code optimization [1,2] is accomplished as part of these macros through conditional transfers and predefined attribute tests, e.g., "Is the needed result in the accumulator?" "Is the argument a constant?" Nested, but not recursive, calling of macros is provided. "Built-in macros" are subroutines which handle some of the more complex

tasks, such as processing LSVs, and perform some intermediate level code optimization e.g., allocation of index registers, and subscript optimization. The output of these routines are macro calls to symbolically defined macros.

The final step is allocation of storage for program variables and constants, LSVs, and the temporary work area.

B.3 Pass 3 - Assembly

The assembly source language is accumulated in a work area on disk. When complete, control is passed to a modified version of the 1800 assembler which accepts its input from this work area. Normal assembler control cards may be included in the input deck to govern the listing of the program and symbol table, to save the object code, etc.

BIBLIOGRAPHY FOR APPENDIX B

1. Arden, Bruce W.; Galler, Bernard A.; Graham, Robert M. "The MAD Definition Facility", Comm. ACM (Aug. 1969) 12, 8, pp. 432-439.

2. Cheatham, T.E., and K. Sattley, "Syntax Directed Compiling", pp. 264-297, in Rosen, Saul, Programming Systems & Languages, McGraw-Hill, 1967.

3. Floyd, Robert W., "Syntactic Analysis and Operator Precedence", J. ACM 10, 7 (July 1963) pp. 316-333.

4. Gries, David, "Use of Transition Matrices in Compiling", Comm. ACM 11, 1 (Jan 1969) pp. 26-34

5. International Business Machines Corp. IBM 1800 Assembler Language IBM Publication C26-5882.

6. ___. IBM 1800 Functional Characteristics IBM Publication A26-5918.

7. ___. IBM 1800 Time-Sharing Executive System Concepts and Techniques IBM Publication C26-3703.

APPENDIX C. RUN TIME ENVIRONMENT IMPLEMENTATION

The Run Time Environment provides facilities for maintaining the cell space data base, invoking the transition routine, receiving and interpreting user commands, and maintaining the displayed description of the cell space state.

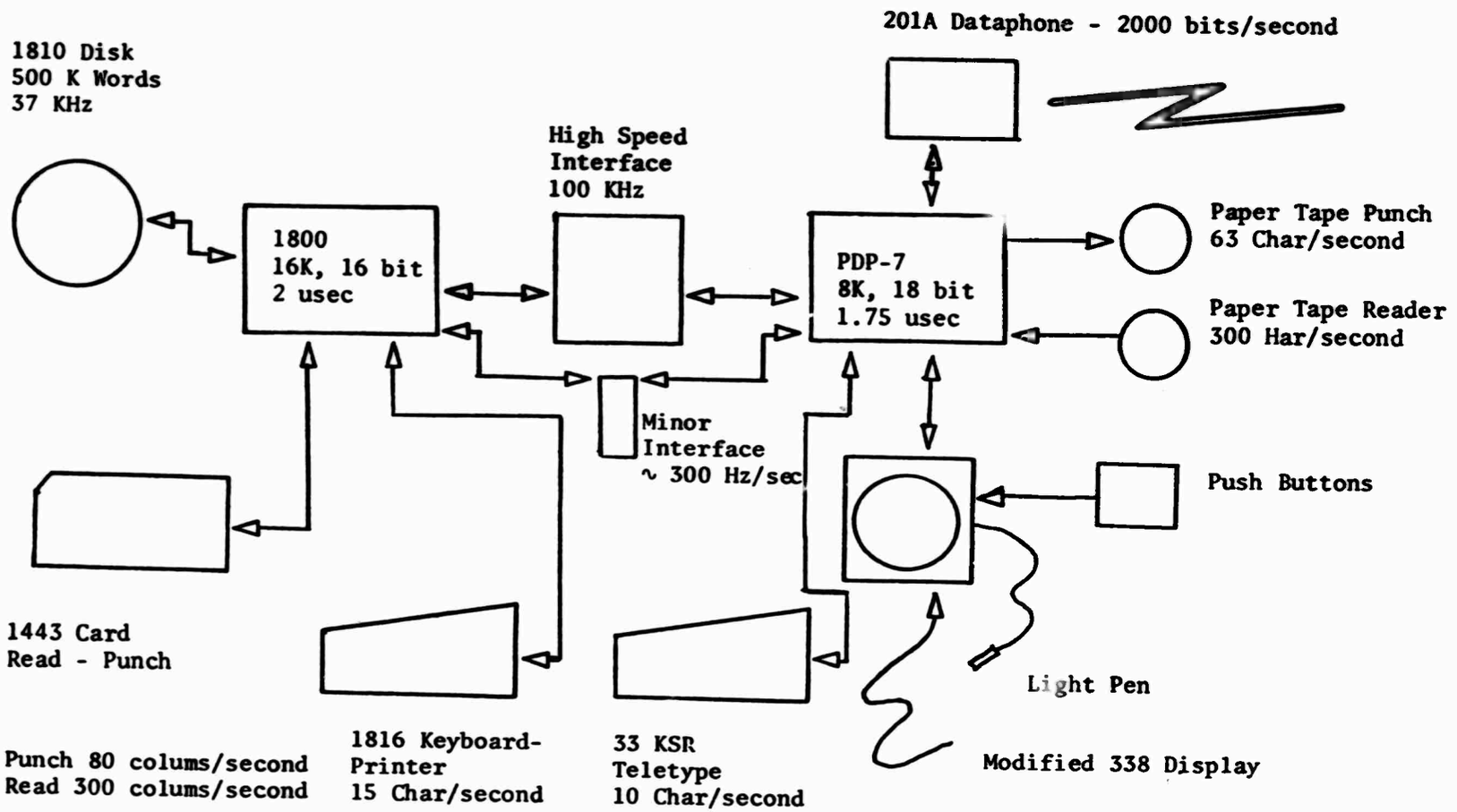
The simulation system is distributed along functional lines between the two computers with the PDP-7 handling user interaction and maintaining the display, and the 1800, with its larger memory and bulk storage, handling the basic computational load. Because two different kinds of CPU's are involved, no attempt at load sharing is possible except by explicit choice of the programmer.

The implementation will be described first in terms of the available hardware. Then the organization of the software on the respective machines will be discussed, and finally the intermachine protocols will be detailed.

C.1 Hardware Configuration

The hardware configuration on which the simulation system is implemented is unusual. See Figure C.1. The hardware consists of two computers, (1) an IBM 1800 with disk and (2) a DEC PDP-7 with CRT display. The display is the DEC 338 display modified by substituting a PDP-7 for the PDP-8 computer of the 338. This specially built "337" is the prototype for DEC's 339 display system.

The two computers are connected by two interfaces. All of the current work has been performed using the slower word-by-word direct program con-



Computer Configuration
Figure C.1

trolled interface because the high-speed interface was not completed. This interface gives a data rate of approximately 300 words (16 bits/word) per second. While acceptable for small simulations, this rate is definitely an annoying factor for larger simulations.

The high-speed interface was designed by this author and John L. Foy, Jr. with the requirements of this computer configuration specifically in mind. This interface features a master-master control organization and unusually flexible formatting of data movement. The interface may be fully controlled by either computer separately or by both cooperatively. Data movement is via a variable length circular shift register providing selectable frame sizes (effective word size). When completed and integrated into this simulation system, the flexibility and high-speed of the interface will help to obviate many of the problems of working with this two computer complex.

C.2 1800 Organization

The 1800 may be considered the "real" simulation machine and is the logical controller of the total computer system. The data base and all user provided routines reside on the 1800. It communicates with the PDP-7 to generate the cellular displays and to accept and act on the operator's commands.

Program listings are not included in this report because 1) they are lengthy, 2) the system is written in assembly language, and 3) the hardware configuration is unique. Rather we have attempted through flowcharts and discussion to make the organization of the system clear enough that others could adapt it if they so desired.

C.2.1 Cell Space Data Structure and Management Routines

The data structure chosen to represent the cell space consists of an ordered set of variable length blocks. Each block represents a row of cells parallel to the X axis. Access to each block is made via three vectors giving the minimum X coordinate of the block, the maximum X coordinate, and the base of the block in memory, respectively. This organization is sometimes called the address table technique. (This "base" is really a displacement from a storage area beginning since two identical data structures must be manipulated.)

The algorithm for computing the base of a cell given its X and Y coordinates is as follows:

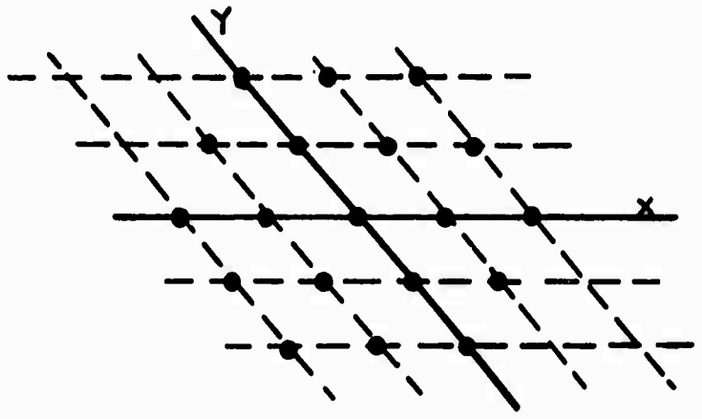
> Let C(X,Y) be the displacement of the cell at coordinates X and Y, and NWPC be the number of words per cell required to represent the state of a single cell in memory; then:
>
> 1. If Y > YMAX or Y < YMIN then C(X,Y) is undefined; else go to 2.
>
> 2. If X > XMAX(Y) or X < XMIN(Y) then C(X,Y) is undefined; else go to 3.
>
> 3. I = Y-YMIN + 1
>
> C(X,Y) = (BASE (I) + X - XMIN(I) + 1) * NWPC

To illustrate, the cellular space with boundary

DEFINEBDRY [-2,-2] [-2,0] [0,2] [2,2] [2,0] [0,2];

may be drawn as shown in Figure C.2. The three arrays, XMIN, XMAX, and BASE for this space are also shown in Figure C.2. The "center" cell with coordinates [X,Y] = [0,0] may be accessed at location C(0,0) = BASE(0-(-2)+1) + 0 - XMIN(0 - (-2) + 1) + 1 = BASE(3) + 0 - XMIN(3)+1 = 7 + 0 - (-2+1 = 10 relative to the base of the space data structure.

DEFINESIZE [-2,2] [-2,0] [0,2] [2,2] [2,0] [0,2]



In the data area of the data structure appears sequentially the rows of the cellular space, beginning with YMIN and XMIN:

C(-2,-2), C(-1,-2), C(0,-2),
C(-2,-1), C(-1,-1), C(0,-1), C(1,-1),
C(-2,0),...
C(0,2), C(1,2), C(2,2)

In the address table appears dimension information and the location of the beginning of each row in the data area:

YMIN: 2
YMAX: 2

I	BASE(I)	XMIN(I)	XMAX(I)
1	0	-2	0
2	3	-2	1
3	7	-2	2
4	12	-1	2
5	16	0	2

Example of Cell Space Data Structure
Figure C.2

Figure C.3 presents the logic of the cell accessing routine, GETST. Note that the default handling of external conditions is trivially handled by a "dummy" external cell procedure that simply returns on being called.

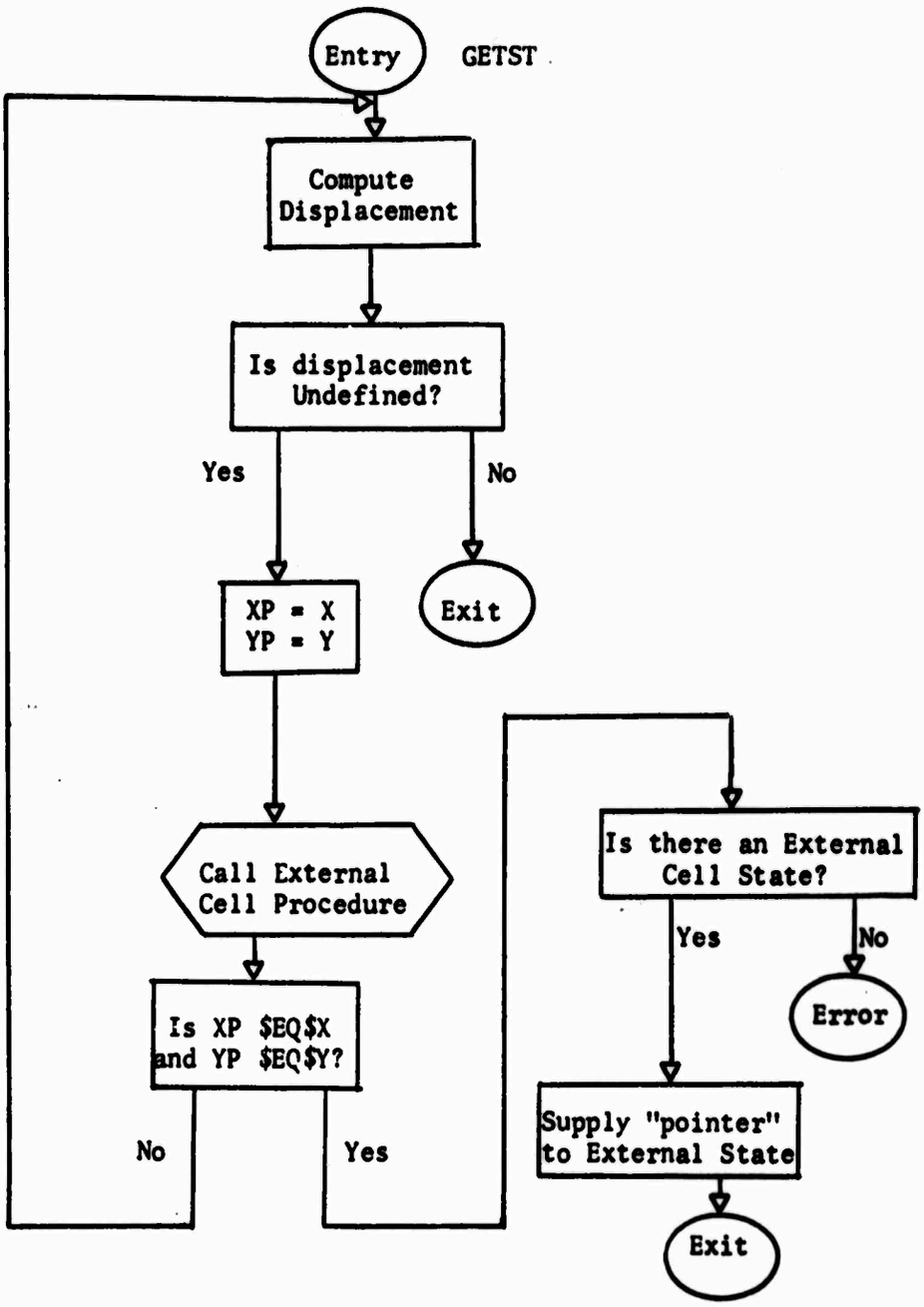
The standard external cell procedure routines are now easily presented in terms of the cell space data structure. The flowcharts are shown in Figure C.4.

All references to the cell space data base are made via the two routines GETST for accessing cells, and PUTST for assigning a value to cells. These routines work from five global parameters: X and Y, coordinates of the relevant cell; OLDGT and OLDPT, base addresses of the arrays to access for GETST and PUTST respectively (these are different during a transition, but will have the same value when changing a state via light pen commands); an index register where the base of the appropriate cell is produced as the value of these routines.

C.2.2 Simulator Organization

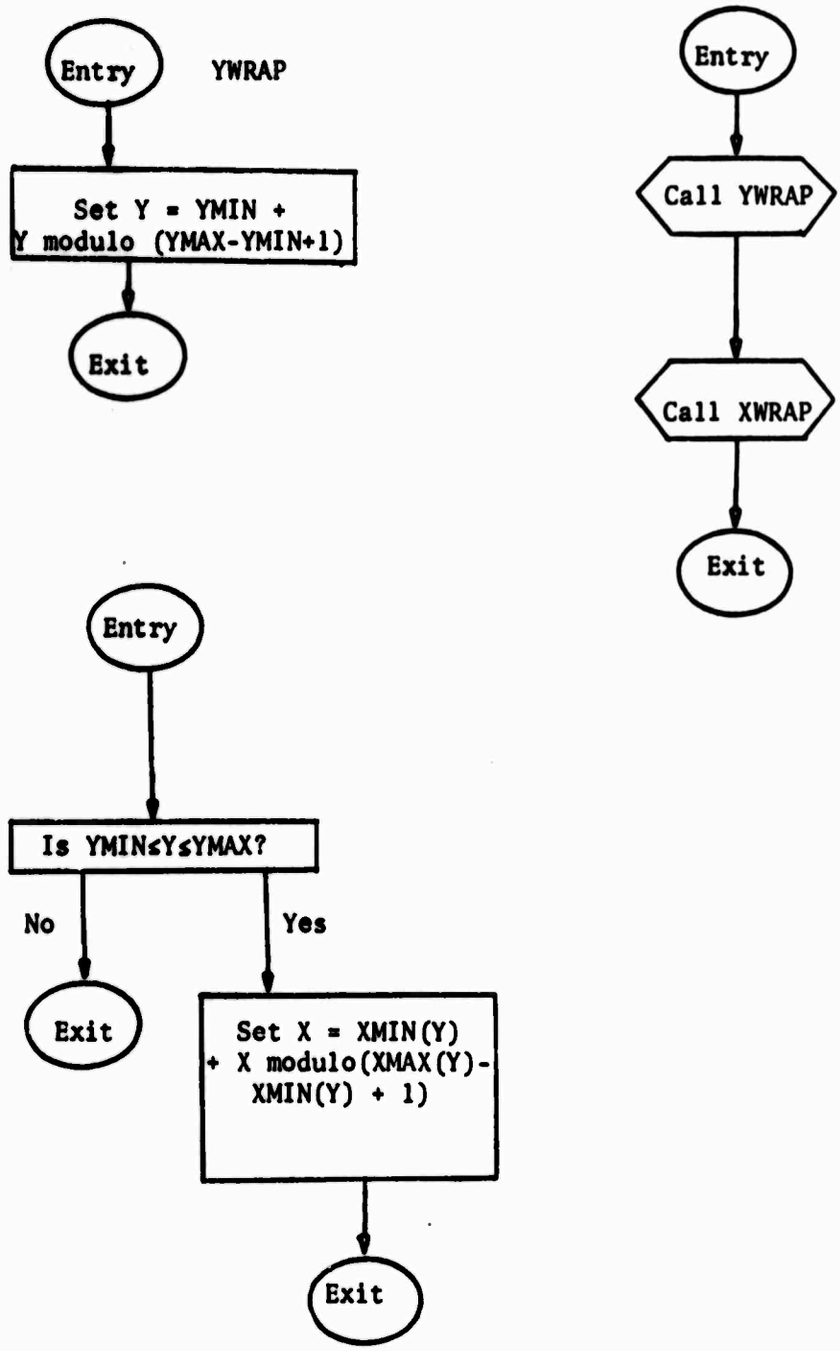
The main routine of the simulation system, CELSP, consists primarily of a simple command dispatcher for directing commands from the PDP-7 to the appropriate routine. The main routine also performs initialization when the system is first loaded. See Figure C.5. Except for computing the state transition, we simply list the major routines and briefly indicate their functions in Figure C.6.

The organization of the transition computing routine is shown in Figure C.7. That flowchart together with the discussion in Section C.2.3 describes this rather complex operation.



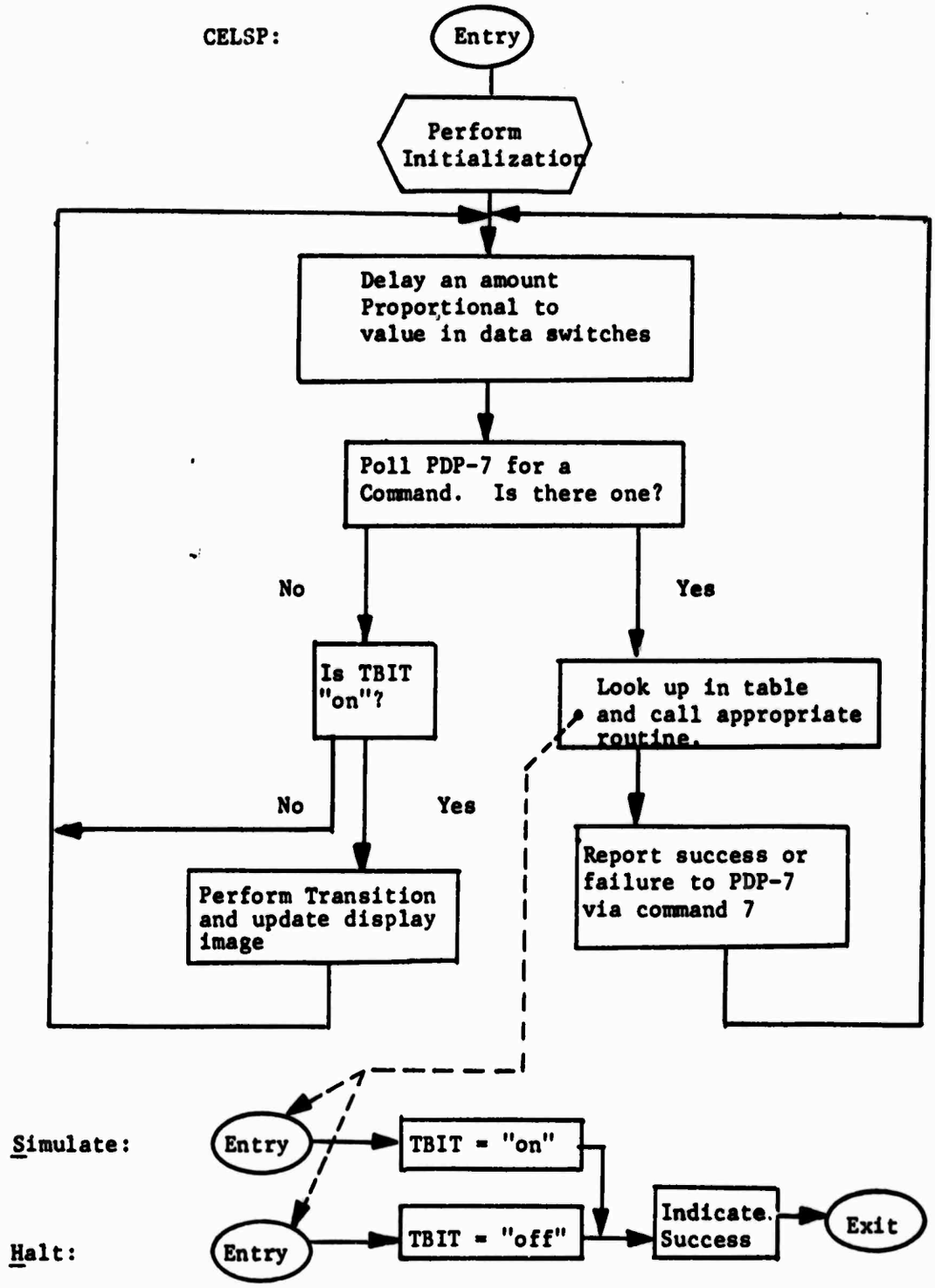
Flow Chart of Cell Accessing Routine

Figure C.3



Flow Charts for Standard External Cell Procedures

Figure C.4



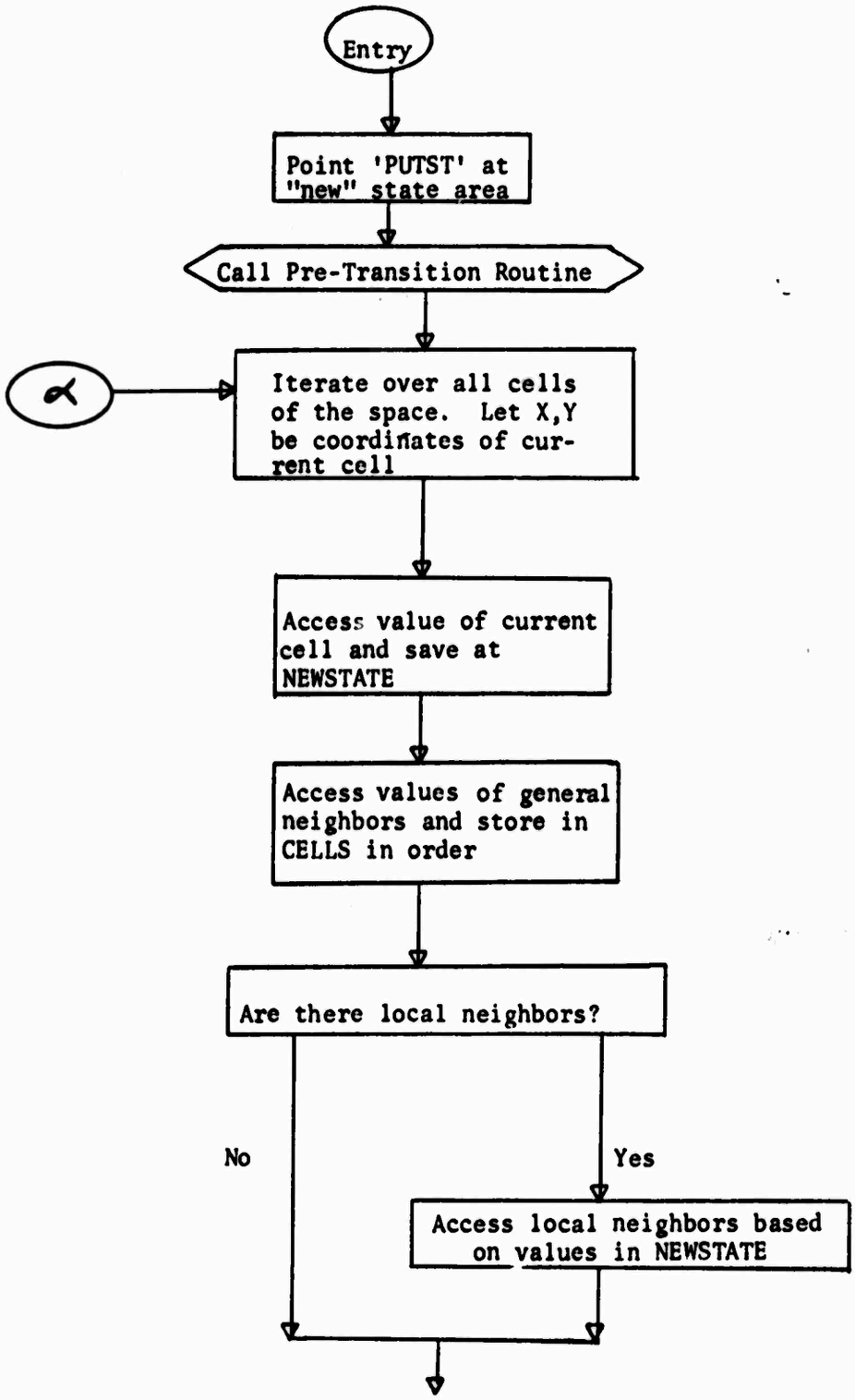
Flow Chart of Main Simulation Routine: CELSP

Figure C.5

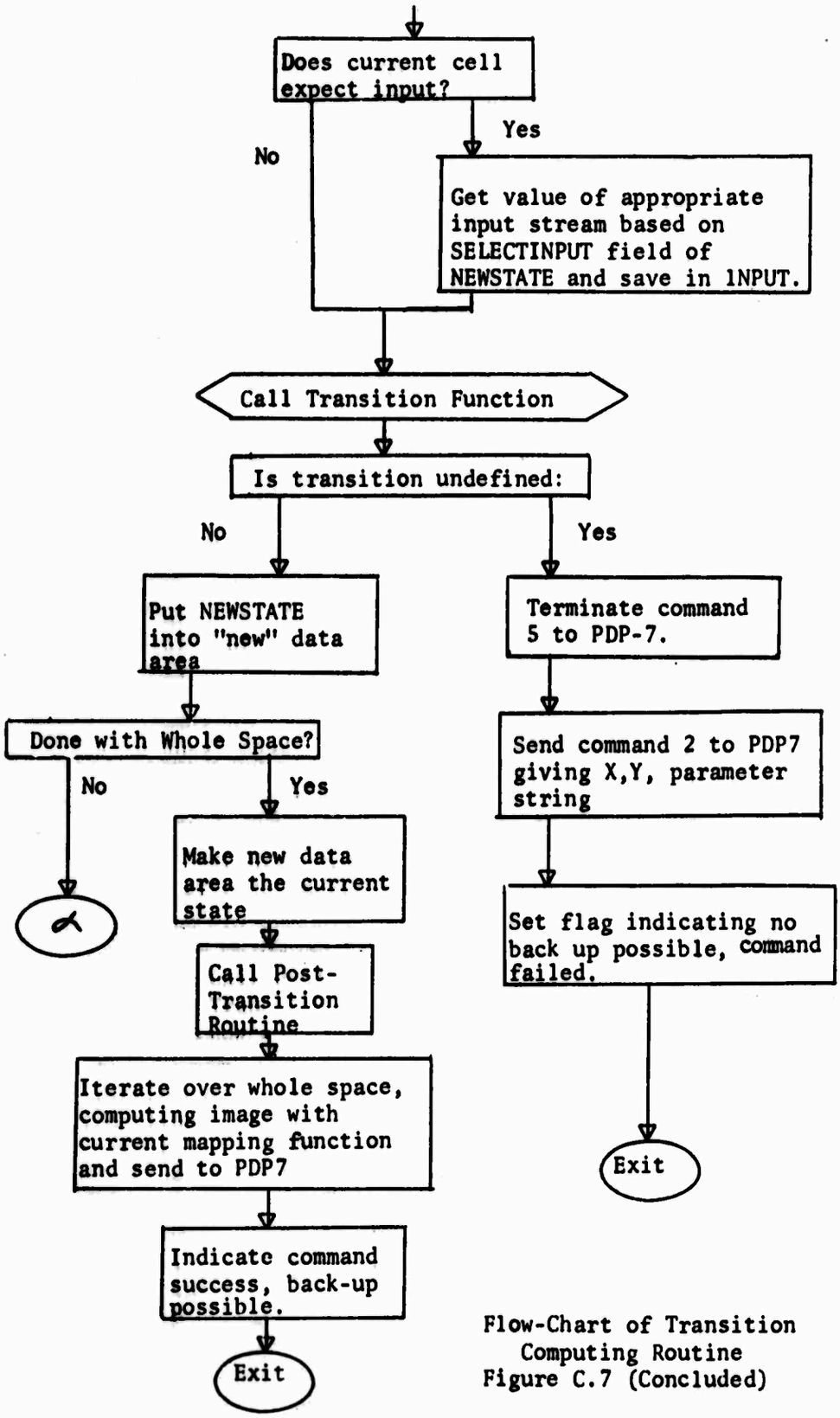
Routine Name	Summary of Function
PARAM, ATDIM, SPEC7	Read and set up user control data in the simulator control area, fill in default conditions where needed; Compute the address table based on boundary spedification and allocate core as needed; Define configuration to PDP-7 (Command 6).
KEEP, REGO	Save and Restore the current state space on an external file.
USER$	Collect parameter string from PDP-7, assemble in USERINPUT area, then call the USER entry.
INPUT	Accept input from PDP-7 and set up control tables for the input stream being defined.
QR	A service routine for PDP-7--converts character string to floating point binary form and returns value.
QR, QFA	Look at the "field table" of the cell data structure and return the displacement and data type of the i^{th} field. QF also reports these to the PDP-7.
MULTI	Accepts X, Y coordinates, field number, and data value from PDP-7 and substitutes into current space state. Return image of the modified cell as computed by the current mapping function.
DEFAL	A collection of default routines which are selectively used if the user does not declare a feature in his program.

Major 1800 Routines and Functions

Figure C.6



Flow-Chart of Transition Computing Routine
Figure C.7



Flow-Chart of Transition Computing Routine
Figure C.7 (Concluded)

Five different commands may be given by the 1800 to the PDP-7. Except for the first, each command initiates a preplanned sequence of data exchanges between the two machines. (This kind of organization was required by the limited capabilities of the available interface.[1]) These commands are:[2]

1. Poll for a PDP-7 command.
2. Initiate processing for an undefined transition.
5. Update the cell space display file as computed by the current mapping function.
6. Define the current cell space configuration to the PDP-7.
7. Report to the PDP-7 the success or failure of the last command accepted from it via command 1.

C.2.3 Structure of Transition Routines

The program produced by the compiler consists of three logical parts: 1) a subroutine call to CELSP with a parameter pointing to 2) a control block containing a description of a simulation to be performed and used to establish linkages for running the simulation and 3) the set of routines (transition function, maps, external cell procedure, etc.) constituting the simulation. This section will describe the control block which provides primary linkage between the compiler and the running environment.

Figure C.8 is an annotated listing of the contents of the control block. It is divided into sections each identified by an integer. For convenience

---

[1] We expect that the new interface will permit more flexible interactive procedures, such as, directly changing control locations in the opposite machine without the necessity of cooperation from the opposite machine.

[2] Numbers correspond to the codes used by the two computers.

```
*
*
*   1. STATE SPACE
*
NWPCU DC      2          NO. WORDS PER CELL UNPACKED
NWPC  DC      2          NO. WORDS PER CELL PACKED
PACK# DC      0          ADR: PACKING ROUTINE
UNPK# DC      0          ADR: UNPACKING ROUTINE
*
*
*   2. NEIGHBORHOOD
*
NNEI  DC      5          NO. OF GENERAL NEIGHBORS
NLIS  DC      $NBR       ADR: NEIGHBOR LIST
NEIG# DC      0          ADR: NEIGHBOR PROCEDURE
INEI# DC      0          ADR: INITIAL NEIGHBOR PROCEDURE
DX    DC      0          COSINE OF AXIS ANGLE
DY    DC      0          SINE OF AXIS ANGLE
*
*
*   3. SIZE OF SIMULATION
*
M     DC      4          NO. OF VERTICES
*(MAXIMUM OF 14 VERTICES)
*KX AND KY ARE LISTS OF VERTEX POINTS FOR X AND Y, RESP.
*    THE XS
KX    DC      0          MUST BE ZERO
      DC      1          "LOWER LEFT"
      DC      1          "UPPER LEFT"
      DC      20         "UPPER RIGHT"
      DC      20         "LOWER RIGHT"
      ORG     KX+16
*    THE YS
KY    DC      0          MUST BE ZERO
      DC      1          "LOWER LEFT"
      DC      20         "UPPER LEFT"
      DC      20         "UPPER RIGHT"
      DC      1          "LOWER RIGHT"
      ORG     KY+16
```

TRANSITION ROUTINE CONTROL BLOCK
FIGURE C.8

```
*
*
*  4. EXTERNAL AND INITIAL STATES
*
ELIS   DC      EXT         ADR: EXTERNAL CELL STATE VALUE
EXTC#  DC      0           ADR: EXTERNAL CELL PROCEDURE
ILIS   DC      0           ADR: INITIAL CELL STATE VALUE
INIS#  DC      0           ADR: INITIAL STATE PROCEDURE
*
*
*  5. TRANSITIONS
*
BEFR#  DC      0           ADR: PRE-TRANSITION ENTRY
AFTR#  DC      0           ADR: POST-TRANSITION ENTRY
TRAN#  DC      MOD8        TRANSITION ROUTINE ENTRY
*
*
*  6. DISPLAY MAPPING ROUTINES
*
MAP#   DC      0           ADR: ENTRY TO MAP 0
       DC      MAP1        ADR: ENTRY
       DC      MAP2        ETC.
       DC      MAP3
       DC      0           #4
       DC      0
       DC      0
       DC      0
       DC      0
       DC      0           #9
*
*
*  7. MISC
*
USER#  DC      USERR       ADR: USER ENTRY
FLDTB  DC      $FLD        ADR: FIELD AND DISPLACEMENT TABLE
CORB#  DC                  COMPILER VERSION
CELNB  DC      0           ADR: LOCAL NEIGHBOR TABLE
INSLT  DC      $INPT       ADR: INPUT SELECT FIELD
```

TRANSITION ROUTINE CONTROL BLOCK
FIGURE C.8 (CONTINUED)

```
*
*    END OF FIXED LENGTH PART
*
$NBR   DC      0          "CENTRAL CELL"
       DC      0
       DC      0          "UP"
       DC      1
       DC      1          "RIGHT"
       DC      0
       DC      -1         "DOWN"
       DC      0
       DC      0          "LEFT"
       DC      -1
*
$FLD   DC      2          TWO FIELDS PER CELL
       DC      0          DISPLACEMENTS: 0 TO FIELD 1,
       DC      1          1 TO FIELD 2.
       DC      2          DATA TYPES: BOTH INTEGERS.
       DC      2
*
$INPT  DC      0          FIRST WORD IS INPUT SELECT FIELD.
*
*    END OF CONTROL BLOCK
*
*      VARIABLE STORAGE AND USERS ROUTINES FOLLOW.
*
```

TRANSITION ROUTINE CONTROL BLOCK
FIGURE C.8 (CONCLUDED)

we shall refer to lines relative to these sections, e.g., line 6.2 is the second line in section 6. The values shown in the Figure are the values that are generated by the compiler for the MOD8 cell space listed in Figure 3.3.

Line 1.1 gives the number of machine words required to represent the state of a cell. Line 1.2 gives the number of words per cell in the unpacked format manipulated by the transition routines. Provision has been made in the run time system (but not supported by the language) to use a different data format within the cell space data structure as compared with that manipulated by the transition function. For example, several boolean variables could conveniently be packed into a single word in the data base but are most conveniently mainpulated on a one-per-word basis by the transition routine. This facility has not been exploited and is not discussed further.

Lines 2.1 and 2.2 define the general neighborhood. Line 2.1 is the number of neighbors and 2.2 is a pointer to a block twice that length giving, in the order declared, the relative X and Y displacements of neighbors.

Lines 3.1 to 3.5 define the boundary of the cell space with the X and Y coordinates listed in order in spearate vectors. The example defines a 25 by 25 square boundary.

The initial and external states are declared in lines 4.1 to 4.4. Line 4.1 is a pointer to the value to be used for external cells, when needed. Line 4.3 is a similiar pointer for the initial state. The pointers are zero if no value is declared. The address of the external cell procedure is given in line 4.2. If zero, default handling is provided.

Line 7.2 is a pointer to a table that provides the data displacements and data types to the run time environment. The first word of the block gives the number of fields of a cell,[1] call this NFPC. Following is a block of length NFPC whose i^{th} entry gives the displacement relative to the base of a cell of the i^{th} field. Next follows another block of NFPC words whose i^{th} entry represents the data type of the i^{th} field, encoded as follows:

BOOLEAN	1
INTEGER	2
REAL	3
all others	0

Line 7.3 contains the version of the compiler that produced this program and is used by the Run Time Environment to provide compatability with older versions while developing and debugging a new version of the compiler.

Local neighbors are indicated by line 7.4 which is a pointer to a local neighbor table. The first word of this table is the number of local neighbors, NLNB. Following is a block of size NLNB whose i_{th} entry is the relative displacement of base of the i^{th} data item of type COORD designating the local neighbor values. For example, if the state of cell was defined as

```
DECLARE STATE BLOCK [INTEGER, COORD, COORD];
DEFINE STATE: CELL;
```

then the following table would be generated:

---

[1] On the 1800, a 16 bit word machine, real numbers are represented by two words of storage. Thus, the number of fields may be less than or equal to, the number of words in line 1.1.

```
DC      2       two local neighbors
DC      1       first COORD is at second word
DC      3       second COORD is at fourth word.
```

If input is declared, then line 7.5 points to a word containing the displacement of the field determining the input stream.

C.3 PDP-7 Software

The PDP-7 code consists primarily of three parallel tasks. These tasks are 1) keyboard command language task, 2) display maintenance task, and 3) PDP-7/1800 communication task. In fact, it is difficult to divide the program quite this cleanly, since control flows among these tasks, sometimes in parallel fashion, and sometimes, serially. Several other service tasks are invoked from time to time for various special purposes.

The keyboard task is more or less the master task. It accepts input either 1) from the display when the command menu is up or 2) from the keyboard at all times. Specific commands may cause the display to be changed and/or information to be passed from PDP-7 to 1800 via the communication task. Commands invoked by this task may accept arguments from the keyboard and are responsible for their own format control and error detection. The 'U' and 'I' commands, for example, do not interact with the 1800 until the entire command input is processed and accepted; then the data is shipped in a "burst".

When sending commands to the 1800 for processing, the keyboard task waits after each command for a success or failure response, (via command 7). If success is reported, the command interpretation continue; if failure is reported, then the current command sequence is aborted, the micro-program

(if any) terminated, an error message given, and processing initialized for new commands.

The display management task is several relatively independent tasks, one for each display image. Once a display is established, it is maintained via interrupts from the display hardware.

For the menu type images, the display is built from a series of button control blocks. Each control block contains: 1) X and Y coordinates at which to place the button on screen, 2) a pointer to the text to comprise the button, 3) a bit indicating if the button is to be light pen sensitive, and 4) if light pen sensitive, a parameter to be saved in a buffer upon a light pen hit. Two special buttons, CANCEL and C.R., either empty the buffer or release the buffer to be read by the command processor, respectively.

A second major component of the display task is the display cells section. The entire cell space output image as produced by the mapping functions is kept in PDP-7 core. A routine called BLANK produces a set of control tables to drive the display to show the portion (if any) of the space logically in the display window. Since the output state data is not changed or edited when moving the display window, the 1800 need never be aware of the current window position or its logical edges.

Light pen hits may invoke special tasks, such as, transmitting cell identity during a MULTI DEFINE operation and handling the updated map value.

The communication task acts like a funnel to control the multiple tasks that may want to interact with the 1800. The communication task itself is controlled by commands from the 1800. One of the commands means in effect

"Give me any command you may have for me." The communication task then enables the data transfer routines to be seized. Since several data transfers in either direction may be involved in command execution, the data transfer routines must be explicitly released by the using routines when an interchange is completed, in order to let any further commands" fall through".

C.4 Communication Protocols

This section describes the communication protocols used between the 1800 and PDP-7 during data exchanges. These protocols are tabulated in Figure C.9. The following conventions are used in this Figure:

There are two columns, with the left column representing data sent from the 1800 to the PDP-7 and the right, the converse. Numerals down the left side of the page designate independent command sequences. Within a given sequence, capital letters are used to designate changes of transfer direction. For example, in the first command the 1800 sends the command and the PDP-7 responds with either a zero (indicating no PDP-7 command at present) or by initiating a PDP-7 command. In the second command, information is passed only to the PDP-7 and no response is involved.

Most commands from the PDP-7 do not involve a command sequence of their own but many result in 1800 initiated commands, such as updating the cell space display.

	1800	PDP7
1.	A. Command 1	
	B.	a. Zero "nothing for you" b. non-zero is command code
	- - - -	
2.	A. Command 2 X and Y coordinates of undefined transition a string of text characters terminated by zero	
	- - - -	
3.	A. Command 5	
	B.	Return 0
	C. Map Number	
	D.	Return 0
	E. Time Step Number	
	F.	Return 0
	G. The cell graphics, one for each cell	
	H.	Return 0
	- - - -	
4.	A. Command 6	
	B.	Return 0
	C. Values of YMIN, YMAX Values of arrays XMIN, XMAX and BASE Values of DXDY (slope of axis) Number of fields per cell End-of-file	
	D.	Return 0

1800/PDP-7 Interactive Protocols
Figure C.9

	1800	PDP7
5.	A. Command 7 Success or failure of previous command	
	- - - -	
6.	A.	0,1,2,3,...,9,N,B,S,H,C,X,E
	- - - -	
7.	A.	Command "U", "I" A string of characters and values End-of-file
	- - - -	
8.	A.	Command "M" (light pen hit) X and Y coordinates of a cell field number value
	B. Display character for current map	
	- - - -	
9.	A.	Command QF Field number
	B. Data type of given field	
	- - - -	
10.		Command QR A string of numerals, period or negative sign terminated by zero.

1800/PDP-7 Interactive Protocols
Figure C.9 (Continued)

1800	PDP7

B. a. If valid conversion:
Non-zero code word
Value (2 words)
b. If not valid:
Zero code

- - - -

11. A. Command K
B. Number of words per cell
Number of fields per cell
Contents of address table
External Cell State value
The Cell Space State
End-of-file

- - - -

12. A. Command R
B. Request one word
C. Number of words per cell
D. Request one word
E. Number of fields per cell
F. Request address table
G. Contents of Address table
H. If same boundry, skip to J´
Else, request external cell
state value
K. External cell state value
L. Request Cell Space State
M. Cell Space State
N. End-of-file

1800/PDP-7 Interactive Protocols
Figure C.9 (Concluded)

BIBLIOGRAPHY FOR APPENDIX C

Brender, R. F., D. R. Frantz, J. L. Foy, Jr., and T. W. Schunior, Specialized System Software for Interacting DEC PDP-7 and IBM 1800, Technical Report 11, Concomp Project, University of Michigan, Ann Arbor, December, 1968.

Brender, R. F., and J. L. Foy, Jr., Flexible High-Speed Interface Between IBM 1800 and DEC PDP-7 Computers, Technical Report 12, Concomp Project, University of Michigan, Ann Arbor, October, 1968.

Digital Equipment Corp. PDP-7 Users Handbook, D.E.C. Publication F-75.

___, Programmed Bufferred Display 338 Programming Manual, D.E.C. Publication DEC-08-G61B-D.

Frantz, D. R., R. F. Brender, and J. L. Foy, Jr., LOCOSS: A Multi-programming Monitor for the DEC PDP-7, Technical Report 10, Concomp Project, University of Michigan, Ann Arbor, October, 1968.

International Business Machines Corp. IBM 1800 Functional Characteristics, IBM Publication No. A26-5918.

___, IBM 1800 Time-Sharing Executive Systems and Techniques, IBM Publication No. C26-3703.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)

THE UNIVERSITY OF MICHIGAN
CONCOMP PROJECT

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

A PROGRAMMING SYSTEM FOR THE SIMULATION OF CELLULAR SPACES

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Report 25

5. AUTHOR(S) (First name, middle initial, last name)

BRENDER, RONALD F.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
January 1970	170	27

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DA-49-083 OSA 3050	Technical Report 25
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Advanced Research Projects Agency

13. ABSTRACT

Regular networks of similar interacting components constitute an important class of models in many disciplines, from automata theory to parallel computer systems to biological systems. Yet, no simulation system provides comprehensive facilities for studying such models conveniently by computer. Such a system is proposed and an implementation exhibited. Careful attention is given to setting forth the guiding considerations in developing the final form of the system for supporting heuristic and interactive exploration of model behavior.

Chapter 1 develops the notions of cellular spaces (regular geometry, neighborhood template, transition function) and reviews models used by von Neumann, Codd, Flanigan, Finley and Holland. Chapter 2 analyses these models and formulates the requirements for building a simulation system suitable for a wide range of cellular models. Chapters 3 and 4 describe a total programming system for simulation. A language is designed that provides novel constructs useful for cellular models. A simulation support system provides on-line monitoring of model behavior on a graph CRT and experimenter interaction with the system via keyboard and lightpen. Chapter 5 discusses several applications developed on the system, and evaluates and summarizes the work. Several appendices detail the implementation.

DD FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification